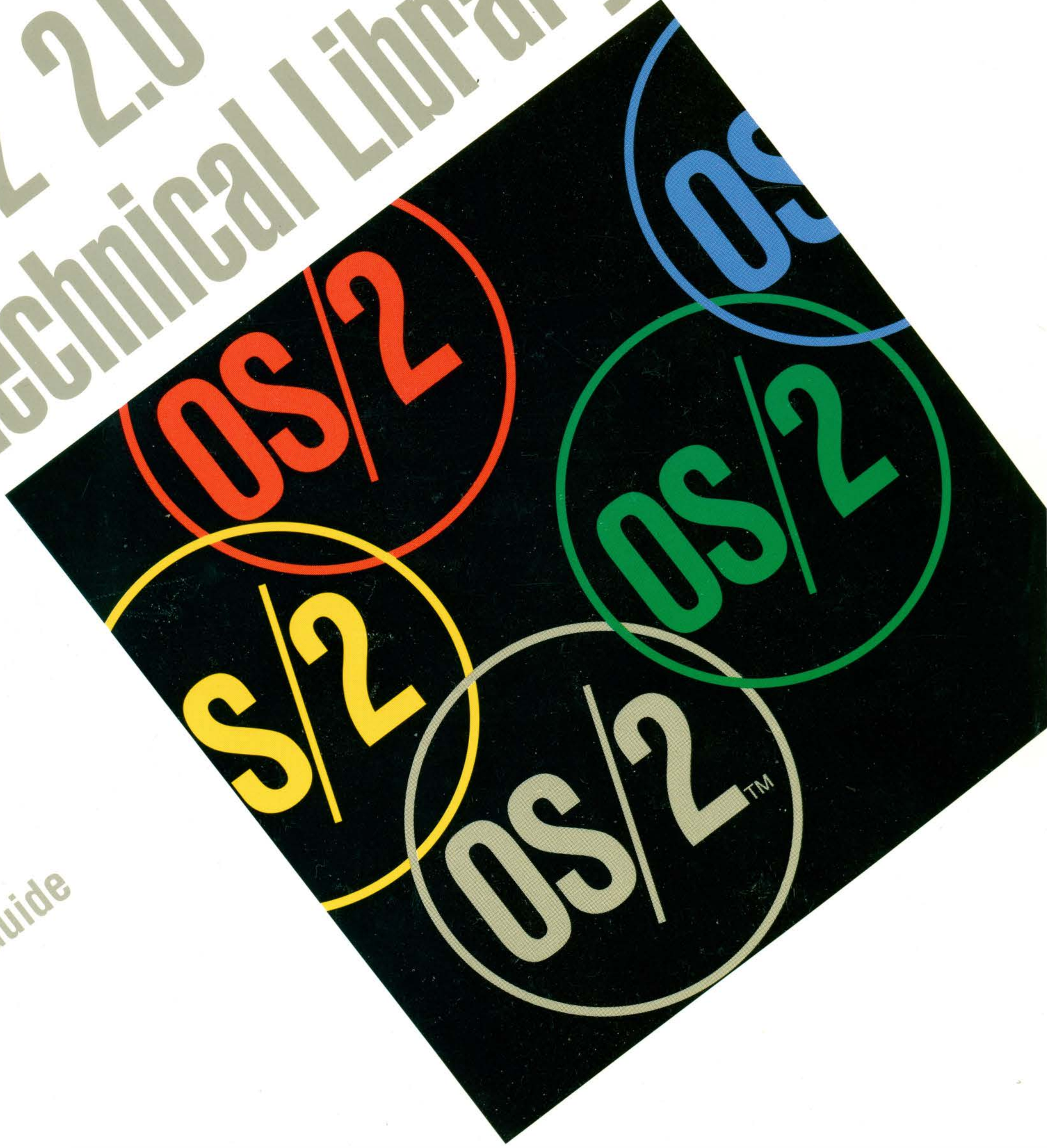
OS/2 2.0 Technical Library



Programming Guide Volume II

Version 2.00

IBM

Programming Family

OS/2 2.0 Technical Library

Programming Guide Volume II

Version 2.00

IBM

Programming Family

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xxiii.

First Edition (March 1992)

Contents

Presentation Manager Window Programming Interface

Figures

Tables

Notices

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

IBM	IBM C/2
Operating System/2	OS/2
Systems Application Architecture	SAA
Personal System/2	PS/2
Common User Access	CUA
Presentation Manager	

The following terms, denoted by a double-asterisk (**) in this publication, are trademarks of other corporations, as follows:

Helvetica	Trademark of Linotype Co.
Times New Roman	Trademark of Monotype Corp.

Double-Byte Character Set (DBCS)

Throughout this publication, you will see reference to specific values for character strings. The values are for single-byte character set (SBCS). If you use the double-byte character set (DBCS), notice that one DBCS character equals two SBCS characters.

Common User Access (CUA) Terminology

For the understanding of the Programming Guide audience, there are instances in this document when the terminology is not compliant with the 1991 CUA User Interface Guidelines. The first occurrence of such instances is noted in text and in appendixes in the back of the book.

About This Book

The three volumes of the *IBM OS/2 2.0 Programming Guide* provide information and code examples to enable you to start writing source code, using the functions in the application programming interface (API) of the OS/2* 2.0 operating system (OS/2). Each volume covers a different facet of the operating system, as follows:

Programming Guide: Volume I—Control Program Programming Interface
Introduces you to the Control Program Programming Interface and describes the functionality provided by the base operating system.

Programming Guide: Volume II—Presentation Manager Window Programming Interface (this book).
Describes the Presentation Manager* (PM) window programming interface. This volume will familiarize you with the windowed, message-based, PM user interface.

Note: Except where noted in text and in the appendixes, this document conforms to the 1991 IBM Systems Application Architecture* (SAA*) Common User Access* (CUA*) guidelines for the new Presentation Manager API functions.

Programming Guide: Volume III—Graphics Programming Interface
Describes the Graphics Programming Interface. This volume provides information on how to prepare graphical output for display and printing.

For complete and comprehensive information about the API, refer to the *OS/2 2.0 Control Program Programming Reference* and the *Presentation Manager Programming Reference—Volumes I, II, and III.*

For information on how to compile and link your programs, refer to the compiler publications for the programming language you are using.

The OS/2 2.0 operating system is a 32-bit system, and this guide is about programming 32-bit applications. (Sixteen-bit applications still are supported by the operating system.)

To illustrate programming with the API, this guide makes extensive use of code fragments. Also, there are sample applications available with the *Developer's Toolkit for OS/2 2.0* (Toolkit). You should familiarize yourself with the operation of each sample from a user's viewpoint. That will help you understand the code in the samples.

Structure of the Books

Each chapter of these books is divided into two sections: *about* the topic and *using* the functions related to that topic. The first section of each chapter provides concepts, terms, and background material; the second section describes the applicable functions and is divided into subsections, each providing information about how to accomplish a specific task. Code fragments are included for most of the functions.

Prerequisite Knowledge

These books are for application designers and programmers who are familiar with the following:

- Information contained in the Application Design Guide
- Information contained in the Control Program and Presentation Manager reference materials
- C Programming Language.

Note: Programming experience on a multitasking operating system also would be helpful.

Presentation Manager Window Programming Interface

Chapter 1. Windows

To most users, a *window* is a rectangular area of the display screen where an application receives input from the user and displays output. This chapter describes the parts of the operating system that enable a Presentation Manager* (PM) application to create and use windows; manage relationships between windows; and size, move, and display windows. An overview of the following topics is presented:

- Types of windows
- Window classes and styles
- Window-creation techniques
- Window messages and message queues
- Methods of window input and output
- Window resources and procedures
- Window identification and modification.

Subsequent chapters present more in-depth descriptions of windows, their advantages and uses, along with example code fragments.

About Windows

The only way a PM application can interact with the user and perform tasks is by way of windows. Each window shares the screen with other windows, including those from other applications. The user employs the mouse and keyboard to interact with the windows and with the applications that own the windows.

Desktop Window and Desktop-Object Window

OS/2* automatically creates the *desktop window* (known as the *workplace* in user terminology) when it starts a PM session.

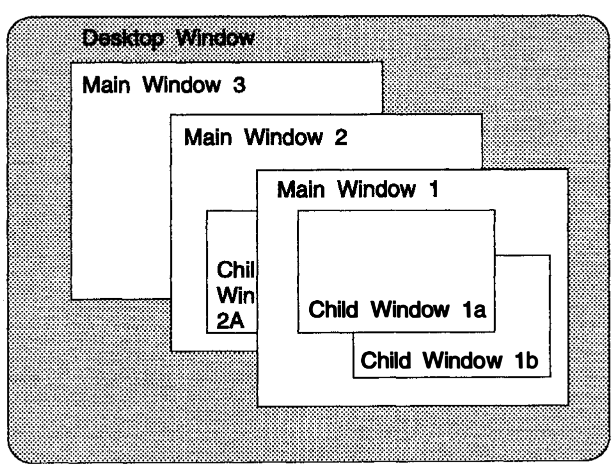


Figure 1-1. Desktop Window Containing Windows of Several Applications

The desktop window paints the background color of the screen and serves as the "progenitor" of all the windows displayed by all PM applications (but not of object windows, which do not require screen display). To make the desktop the parent in the WinCreateStdWindow function, you specify HWND_DESKTOP.

The windows immediately below the desktop are called *main* or *top-level* windows; these are called *primary windows* in user terminology. Every PM application creates at least one window to serve as the main window for that application. Most applications also create many other windows, directly or indirectly, to perform tasks related to the main window.

Each window helps display output and receive input from the user. Figure 1-1 on page 1-1 shows the desktop window containing windows of several applications. Notice that the main windows can overlap one another. (At times, it is possible for a main window to be completely hidden.) Operations in one main window normally do not affect the other main windows.

The *desktop-object window* is like a desktop window that is never displayed; it serves as the base window to coordinate the activity of an application's object windows. The desktop-object window cannot display windows nor process keyboard and mouse input. The primary purpose of the desktop-object window is to enable you to create windows that need not respond to messages at the same rate as the user interface.

Window Relationships

Window relationships define how windows interact with each other—on the screen and through messages. There are parent-child window relationships and window-owner relationships.

The *parent-child relationship* determines where and how windows appear when drawn on the screen. It also determines what happens to a window when a related window is destroyed or hidden. The parent-child rules apply to all windows at all times and cannot be modified.

Ownership determines how windows communicate using messages. Cooperating windows define and carry out their rules of ownership. Although some windows (such as windows of the preregistered public window class, WC_FRAME) have very complex rules of ownership, the application usually defines the ownership rules.

Figure 1-2 represents the logical relationship of the windows in two applications.

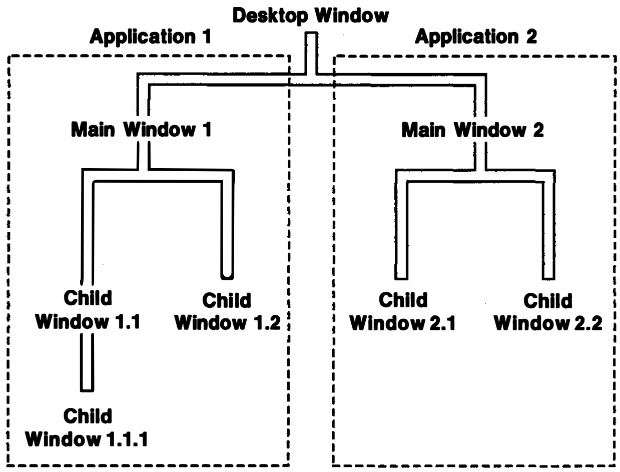


Figure 1-2. Typical Window Relationships

Parent-Child Relationship

Most windows have a *parent window*. (The exceptions are the desktop and desktop-object windows, which the system creates at system startup.) An application specifies the parent when it creates a window; then, the system uses the parent to determine where and how to draw any new windows, as well as when to *destroy* the windows (free all associated resources and remove the windows from the screen).

A *child window* is drawn relative to its parent. The coordinates given to specify the position of a window's lower-left corner are relative to the lower-left corner of its parent. For example, a main window (child of the desktop) is drawn relative to the lower-left corner of the screen (the desktop window's lower-left corner).

All main windows are *siblings* because they share a common parent, the desktop window. Because sibling windows can overlap, an application or a user arranges the windows, one behind another (like a stack of papers on a desk), in the desired viewing order (called *z-order*) as illustrated in Figure 1-1 on page 1-1. Z-order uses the desktop as a reference point for a "three-dimensional" ranking of the overlapping windows: the topmost window has the highest ranking, while the window at the bottom of the stack has the lowest ranking. The parent of the sibling windows is always at the bottom of the z-order.

Figure 1-3 illustrates the hierarchy of such an arrangement.

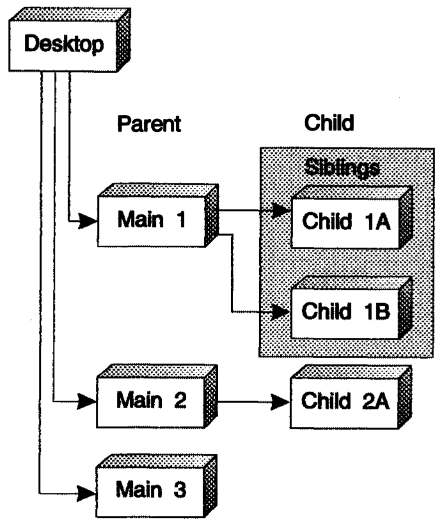


Figure 1-3. Window Hierarchy

Although PM *supports* z-order, it does not *enforce* the expected appearance unless you specify the CS_CLIPCHILDREN or CS_CLIPSIBLINGS styles. No part of a child window ever appears outside the borders of its parent. If an application creates a window that is larger than its parent, or positions a window so that some or all of it extends beyond the borders of the parent, the extended portion of the child window is not drawn.

An application can use the WS_CLIPCHILDREN or WS_CLIPSIBLINGS styles to remove from a window's *clipping area* (the area in which the window can paint) the area occupied by its child or sibling windows. For example, an application can use these styles to prevent a window from painting over a child or sibling window containing a complex graphic that would be time-consuming to redraw.

When a window is minimized, hidden, or destroyed, all of its children are hidden, minimized, or destroyed as well. The order of destruction is always such that every window is destroyed before its parent. The window-destruction sequence starts at the bottom of descendancy so that all related windows can be cleaned up; the last one to go is the window you asked to be destroyed. The final PM task in a window-destruction sequence is to send a WM_DESTROY message to that window, so it has one last chance to release any resources it has allocated and may still be holding.

Every window has only one parent, but can have any number of children. Referring back to Figure 1-3, any window in this tree is said to be a *descendant* of any window appearing above it in the branch, and an *ancestor* of any window appearing below it. There are two special cases, of course: the window immediately above is called the window's *parent*, and any window immediately below it is called its *child*. An application can change a window's parent window at any time by using the WinSetParent function. Changing the parent window also changes where and how the child window is drawn. The system displays the child within the borders of the new parent and draws the window according to the styles specified for the new parent.

Ownership

Any window can have an *owner window*. Typically, an application uses ownership to establish a connection between windows so that they can perform useful tasks together. For example, the title bar in an application's main window is owned by the frame window; but, together, the user can move the entire main window by clicking the mouse in the title bar and dragging. An application can set the owner window when it creates the window or at a later time.

Ownership establishes a relationship between windows that is independent of the parent-child relationship. While there are few predefined rules for owner- and owned-window interaction, a window *always* notifies its owner of anything considered a *significant event*.

The preregistered public window classes provided by OS/2* recognize ownership. Control windows of classes such as WC_TITLEBAR and WC_SCROLLBAR, notify their owners of events; frame windows, of class WC_FRAME, receive and process notification messages from the control windows they own. For example, a title-bar control sends a notification message to its owner when it receives a mouse click. If the owner is a frame window, it receives the notification message and prepares to move itself and its children.

Owner and owned windows must be created by the same thread; that is, they must belong to the same message queue. Because ownership is independent of the parent-child relationship, the owner and owned windows do not have to be descendants of the same parent window. However, this can affect how windows are destroyed. Destroying an owner window does not necessarily destroy an owned window. Except for frame windows, an application that needs to destroy an owned window that is not a descendant of the owner window must do so explicitly.

Frame windows sometimes own windows that are not descendants but, instead, are siblings. A frame window has the following special ownership properties:

- When the frame window is destroyed, it destroys all of the windows it owns, even if they are not descendants.
- When a frame window moves, the windows it owns move also. Owned windows that are not descendants maintain their positions, relative to the upper-left (not the usual lower-left) corner of the owner window. An owned window with the style FS_NOMOVEWITHOWNER does not move.
- When the frame window changes its position in the z-order, it changes the z-order of all the windows it owns.
- When the frame window is minimized or hidden, it hides all the windows it owns. Owned windows hidden this way are restored when the frame window is restored.

If an application needs this type of special processing for its own window classes, it must provide that support in the window procedures for those classes.

Object Windows

Any descendant of the desktop-object window is called an *object window*. Typically, an application uses an object window to provide services for another window. For example, an application can use an object window to manage a shared database. In this way, a window can obtain information from the shared database by sending a message to and receiving a reply from the object window.

Only two system-defined messages are available to an object window—

WM_CREATE and WM_DESTROY—but the object window enables the user to implement a set of user-defined messages. The window procedure for an object window does not have to process paint messages or user input. The object window processes only messages that affect the data belonging to the object.

HWND_OBJECT is the only identifier needed to create an object window. It is very unwise to create descendants of HWND_OBJECT in the same thread that creates descendants of HWND_DESKTOP: this causes the system to hang up or, at the very least, behave slowly. Object windows, sometimes referred to as *orphan* windows, require no owner.

The rules for parent-child and ownership relationships also apply to object windows. In particular, changing the parent window of an object window to the desktop window, or to a descendant of the desktop window, causes the system to display the object window if the object window has the WS_VISIBLE style.

Application Windows

An application can use several types of *secondary* windows: frame windows, client windows, control windows, dialog windows, message boxes, and menus. Typically, an application's main window consists of several of these windows acting as one. Figure 1-4 shows an example of a main window and its secondary windows.

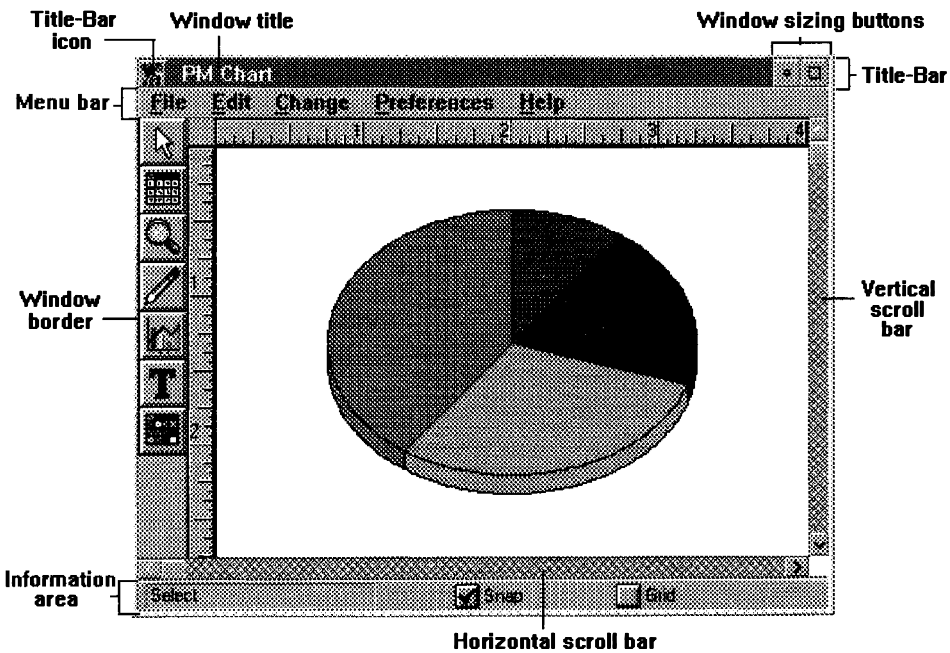


Figure 1-4. Main Window with Secondary Windows

A *frame window* is a window that an application uses as the base when constructing a main window or other composite window, such as a dialog window or message box. (A *composite window* is a collection of windows that interact with one another and are kept together as a unit.) A frame window provides basic features, such as borders and a menu bar. Frame windows have a set of resources associated with them. These include icons, menus, and accelerators (*shortcut keys* to the user), which, typically, are defined in an application's resource file.

A *dialog window* is a frame window that contains one or more control windows. Dialog windows are used almost exclusively for prompting the user for input. An

application usually creates a dialog window when it needs additional information to complete a command. The application destroys the dialog window after the user has provided the requested information.

A *message box* is a frame window that an application uses to display a note, caution, or warning to the user. For instance, an application can use a message box to inform the user of a problem that the application encountered while performing a task.

A *client window* is the window in which the application displays the current document or data. For example, a desktop-publishing application displays the current page of a document in a client window. Most applications create at least one client window. The application must provide a function, called a *window procedure*, to process input to the client window and to display output.

A *control window* is a window used in conjunction with another window to perform useful tasks, such as displaying a menu or scrolling information in a client window. The operating system provides several predefined control-window classes that an application can use to create control windows. Control windows include buttons, entry fields, list boxes, combination boxes, menus, scroll bars, static text, and title bars.

A *menu* is a control window that presents a list of commands and other menus to the user. Using a mouse or the keyboard, the user can select a task; the application then performs the selected task.

Window Input and Output

The user directs input data to windows from a mouse and the keyboard. Keyboard input goes to the window with *input focus*, and, normally, mouse input goes to the window under the mouse pointer.

Windows also are places to display output data. PM uses windows to display text and graphics on the screen and to process input from the mouse and keyboard. Windows provide the same input and output capabilities as a virtual graphics terminal without having direct control of the hardware.

An application is responsible for painting the data for the window classes it registers and creates. This data can be graphics text or pictures or fixed-size alphanumeric text. Normally it is not necessary for the application to paint the system-provided window classes; the OS/2 window procedures for those window classes do the painting.

Active Window and Focus Window

All frame-window ancestors of the input focus window are said to be *active*, meaning that the user interacts with them. The active window usually is the topmost main window, which is positioned above all other top-level windows on the screen. The active window is indicated by some form of highlighting. For example, a highlighted title bar shows that a standard frame window is active; an active dialog window has a highlighted border. These types of highlighting ensure that the user can see the window that is accepting input.

A main window (or one of its child windows) is activated by using a mouse or the keyboard. When a window is activated, it receives a WM_ACTIVATE message with its first parameter set to TRUE. When it is deactivated, it receives a WM_ACTIVATE message with its first parameter set to FALSE. Figure 1-5 on page 1-8 illustrates user interaction with a window.

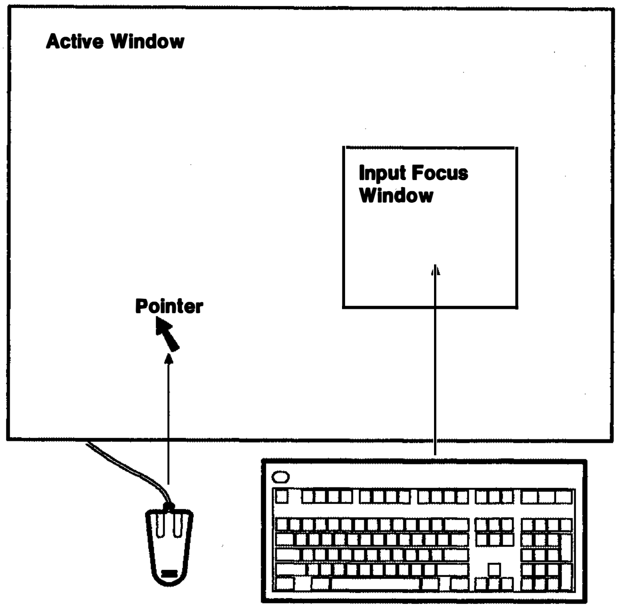


Figure 1-5. User Input to a Window

The *focus window* can be the active window or one of its descendant windows. The user can change the *input focus* the same way active windows are changed—by mouse or keyboard. However, the application has more control over the input focus. For example, in a window containing several text entry fields, the tab keys can move the input focus from one input field to another. A WM_SETFOCUS message is sent to the window procedure when a window is gaining or losing the input focus. The WinQueryFocus function tells the user which window has the input focus.

Messages

Messages are a fundamental part of the operating system. PM applications use messages to communicate with the operating system and one another. The system uses messages to communicate with applications to ensure concurrent running and sharing of devices. Typically, a message notifies the receiving application that an *event* has occurred. The operating system identifies the appropriate application window to receive a message by the window handle included in the message. Sources of events that cause messages to be issued to applications are the user, the operating system, the application, or another application.

The User: Mouse or keyboard input to an application window causes the operating system to direct messages to that window.

The Operating System: Managing the application windows on the screen, the operating system issues messages to the windows, usually as an indirect result of user interaction. These messages enable the system to work in a uniform and well-ordered manner. For example, where several application windows overlap, and the user terminates an application so that its window disappears, the operating system issues messages to the underlying application windows so that they can repaint themselves.

The Application: An event can occur in the application to which another part of that application should respond; for example, when the contents of its window no longer accurately reflect the status of the application. The application can define its own messages outside the range of system-defined messages to communicate such events.

Another Application: Communication with other applications through the operating system ensures cooperative use of the system; it even can be used to exchange data. For example, an arithmetic application can supply the results of a lengthy calculation to a business graphics application.

Enabled and Disabled Windows

An application uses the WinEnableWindow function to enable or disable window input. By default, a window is enabled when it is created. However, an application can disable a newly created window.

An application usually disables a window to prevent the user from using the window. For example, an application might disable a push button in a dialog window. Enabling a window restores normal input; an application can enable a disabled window at any time.

When an application uses the WinEnableWindow function to disable an existing window, that window also loses keyboard focus. WinEnableWindow sets the keyboard focus to NULL, which means that no window has the focus. If a child window or other descendant window has the keyboard focus, it loses the focus when the parent window is disabled.

An application can determine whether a window is enabled by calling WinIsWindowEnabled.

System-Modal Window

An application can designate a *system-modal window*: a window that receives all keyboard and mouse input, effectively disabling all other windows. The user must respond to the system-modal window before continuing work in other windows. An application sets and clears the system-modal window by using the WinSetSysModalWindow function.

Because system-modal windows have absolute control of input, you must be careful when using them in your applications. Ideally, an application uses a system-modal window only when there is danger of losing data if the user does not respond to a problem immediately.

Although an application can destroy a system-modal window, the new active window then becomes a system-modal window. An application can make another window active while the first system-modal window exists. But again, the new active window will become the system-modal window. In general, once a system-modal window is set, it continues to exist in the PM session until the application explicitly clears it.

Window Creation

Before any thread in an application can create windows, it must:

1. Call WinInitialize to create an anchor block
2. Call WinCreateMsgQueue to create a message queue for the thread.

Then, it can create one or more windows by calling one of the window-creation functions, such as WinCreateWindow.

The window-creation functions require that the following information be supplied in some form:

- Class
- Styles

- Name
- Parent window
- Position relative to the parent window
- Position relative to any sibling windows (z-order)
- Dimensions
- Owner window
- Identifier
- Class-specific data
- Resources.

Every window belongs to a *window class* that defines that window's appearance and behavior. The chief component of the window class is the *window procedure*. The window procedure is the function that receives and processes all messages sent to the window.

Every window has a *style*. The window style specifies aspects of a window's appearance and behavior that are not specified by the window's class. For example, the WC_FRAME class always creates a frame window, but the FS_BORDER, FS_DLGBORDER, and FS_SIZEBORDER styles determine the style of a frame window's border. A few window styles apply to all windows, but most apply only to windows of specific window classes. The window procedure for a given class interprets the style and allows an application to adapt a window of a given class for a special circumstance. For example, an application can give a window the style WS_SYNCPAINT to cause it to be painted immediately whenever any portion of the window becomes invalid. Normally, a window is painted only if there are no messages waiting in the message queue.

A window can have a text string associated with it. Typically, the window text is displayed in the window or in a title bar. The class of window determines whether the window displays the text and, if so, where the text appears within the window.

Every window except the desktop window and desktop-object window has a *parent window*. The parent provides the coordinate system used to position the window and also affects aspects of a window's appearance. For example, when the parent window is minimized, hidden, or destroyed, the parent's child windows are minimized, hidden, or destroyed also.

Every window has a screen position, size, and z-order position. The *screen position* is the location of the window's lower-left corner, relative to the lower-left corner of its parent window. A window's size is its width and height, measured in pels. A window's *z-order position* is the position of the window in the order of overlapping windows. This viewing order is oriented along an imaginary axis, the *z* axis, extending outward from the screen. The window at the top of the z-order overlaps all *sibling* windows (that is, windows having the same parent window). A window at the bottom of the z-order is overlapped by all sibling windows. An application sets a window's z-order position by placing it behind a given sibling window or at the top or bottom of the z-order of the windows.

A window can own, or be owned by, another window. The owner-owned relationship affects how messages are sent between windows, allowing an application to create combinations of windows that work together. A window issues messages about its state to its owner window; the owner window issues messages back about what action to perform next.

The *window handle* is a unique number across the system that is totally unambiguous—it identifies one particular window in the system and is assigned by

the system. A *window identifier* is analogous to a "given" name in family relationships—the only requirement is that the name be unique among siblings.

A window can have class-specific data that further defines how the window appears and behaves when it is created. The system passes the class-specific data to the window procedure, which then applies the data to the new window.

Window-Creation Functions

The basic window-creation function is WinCreateWindow. This function uses information about a window's class, style, size, and position to create a new window. All other window-creation functions, such as WinCreateStdWindow and WinCreateDlg, supply some of this information by default and create windows of a specific class or style.

Although the WinCreateWindow function provides the most direct means of creating a window, most applications do not use it. Instead, they often use the WinCreateStdWindow function to create a main window and the WinDlgBox or WinCreateDlg functions to create dialog windows.

The WinCreateMenu, WinLoadMenu, WinLoadDlg, WinMessageBox, and WinCreateFrameControls functions also create windows. Each of these functions substitutes for one or more required calls to WinCreateWindow to create a given window. For example, an application can create a frame window, one or more control windows, and a client window in a single call to WinCreateStdWindow.

Window-Creation Messages

While creating a window, the system sends messages to that window's window procedure. The window procedure receives a WM_CREATE message, saying that the window is being created. The window also receives a WM_ADJUSTWINDOWPOS message, specifying the initial size and position of the window being created. This message lets the window procedure adjust the size and position of the window before the window is displayed.

The system also sends other messages while creating a window; the number and order of these messages depend on the class and style of the window and the function used to create it.

Window Classes

Each window of a specific window class uses the window procedure associated with that class. An application can create one or more windows that belong to the same window class. Because each window of the same class is processed by the same window procedure, they all behave the same way. Since many windows can result from one window procedure, coding overhead is greatly reduced. There are two types of window classes: public and private.

Public Window Classes

A *public window class* is one that has a reentrant window procedure that is registered and resides in a dynamic link library (DLL); it can be used by any process in the system to create windows. The operating system provides several preregistered public window classes. You can specify the system-provided window classes by using the symbolic identifiers that have the prefix WC_, as shown in the following table:

Table 1-1. Window Classes	
Class Name	**Description**
WC_BUTTON	Consists of buttons and boxes the user can select by clicking the pointing device or using the keyboard.
WC_CONTAINER	Creates a control for the user to group objects in a logical manner. A container can display those objects in various formats or views. The container control supports drag and drop so the user can place information in a container by simply dragging and dropping.
WC_ENTRYFIELD	Consists of a single line of text that the user can edit.
WC_FRAME	A window class that can contain child windows of many of the other window classes.
WC_LISTBOX	Presents a list of text items from which the user can make selections.
WC_MENU	Presents a list of items that can be displayed horizontally as menu bars, or vertically as pull-down menus. Menus usually are used to provide a command interface to applications.
WC_NOTEBOOK	Creates a control for the user that is displayed as a number of pages. The top page is visible, and the others are hidden, with their presence being indicated by a visible edge on each of the back pages.
WC_SCROLLBAR	Lets the user scroll the contents of an associated window.
WC_SLIDER	Creates a control that is usable for producing approximate (analog) values or properties. Scroll bars were used for this function in the past, but the slider provides a more flexible method of achieving the same result, with less programming effort.
WC_SPINBUTTON	Creates a control that presents itself to the user as a scrollable ring of choices, giving the user quick access to the data. The user is presented only one item at a time, so the spin button should be used with data that is intuitively related.
WC_STATIC	Simple display items that do not respond to keyboard or pointing device events.
WC_TITLEBAR	Displays the window title or caption and lets the user move the window's owner.
WC_VALUESET	Creates a control similar in function to the radio buttons but provides additional flexibility to display graphical, textual, and numeric formats. The values set with this control are mutually exclusive.

With the exception of WC_FRAME, the system-provided window classes are known as *control window classes*, because they give the user an easy means of controlling specific types of interaction. For example, the WC_BUTTON class allows single or multiple selections. These windows conform to the IBM* Systems Application Architecture* (SAA*) Common User Access* (CUA*) definition. They are designed specifically to provide function that meets the needs for a graphics-based standard user interface. The code fragments provided in this guide make extensive use of the system window classes.

Private Window Classes

A *private window class* is one that an application registers for its own use; it is available only to the process that registers it. The application-provided window procedure for a private window class resides either in the application's executable files or in a DLL file. A private window class is deleted when its registering process is terminated.

Window Styles

A window can have a combination of styles; an application can combine styles by using the bitwise inclusive OR operator. An application usually sets the window styles when it creates the window. The OS/2 operating system provides several standard window styles that apply to all windows. It also provides many styles for the predefined frame and control windows. The frame and control styles are unique to each predefined window class and can be used only for windows of the corresponding class.

Initially, the styles of the window class used to create the window determine the styles of the new window. For example, if the window class has the style CS_SYNCPAINT, all windows created using that class, by default, will have the window style WS_SYNCPAINT.

The OS/2 operating system has the following standard window styles:

Table 1-2 (Page 1 of 2). Standard Window Styles

Style Name	Description
WS_CLIPCHILDREN	Prevents a window from painting over its child windows. This style increases the time necessary to calculate the visible region. This style is usually not necessary, because if the parent and child windows overlap and both are invalidated, the system draws the parent window before drawing the child window. If the child window is invalidated independently of the parent window, the system redraws only the child window. If the update region of the parent window does not intersect the child window, drawing the parent window causes the child window to be redrawn. This style is useful to prevent a child window that contains a complex graphic from being redrawn unnecessarily. WS_CLIPCHILDREN is an absolute requirement if a window with children ever performs output in response to any message other than WM_PAINT. Only WM_PAINT processing is synchronized such that the children will get their messages after the parent.
WS_CLIPSIBLINGS	Prevents a window from painting over its sibling windows. This style protects sibling windows but increases the time necessary to calculate the visible region. This style is appropriate for windows that overlap and that have the same parent window.
WS_DISABLED	Used by an application to disable a window. It is up to the window to recognize this style and reject input.
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the ARROW keys. All controls defined after the control with the WS_GROUP style belong to the same group. The next control with the WS_GROUP style ends the first group and starts a new group.
WS_MAXIMIZED	Enlarges a window to the maximum size.
WS_MINIMIZED	Reduces a window to the size of an icon.

Style Name	Description
WS_PARENTCLIP	Extends a window's visible region to include that of its parent window. This style simplifies the calculation of the child window's visible region but is potentially dangerous, because the parent window's visible region is usually larger than the child window.
WS_SAVEBITS	Saves the screen area under a window as a bit map. When the user hides or moves the window, the system restores the image by copying the bits; there is no need to add the area to the uncovered window's update region. The style can improve system performance but also can consume a great deal of memory. It is recommended only for transient windows, such as menus and dialog windows, not for main application windows.
WS_SYNCPAINT	Causes a window to receive WM_PAINT messages immediately after a part of the window becomes invalid. Without this style, the window receives WM_PAINT messages only if no other message is waiting to be processed.
WS_TABSTOP	Specifies one of any number of controls through which the user can move by tabbing. Pressing the TAB key moves the keyboard focus to the next control that has the WS_TABSTOP style.
WS_VISIBLE	Makes a window visible. The operating system draws the window on the screen unless overlapping windows completely obscure it. Windows without this style are hidden. If overlapping windows completely obscure the window, the window is still considered visible. (*Visibility* means that the operating system draws the window if it can.)

Table 1-2 (Page 2 of 2). Standard Window Styles

Window Handles

After creating a window, the creation function returns a window handle that uniquely identifies the window. An application can use this handle to direct the action of functions to the window. Window handles have the data type HWND; applications must use this data type when declaring variables that hold window handles.

There are special constants that an application can use instead of a window handle in certain functions. For example, an application can use HWND_DESKTOP in the WinCreateWindow function to specify the desktop window as the new window's parent. Similarly, HWND_OBJECT represents the desktop-object window. HWND_TOP and HWND_BOTTOM represent the top and bottom positions relative to the z-order position of a window.

Although the NULL constant is not a window handle, an application can use it in some functions to specify that no window is affected. For example, an application can use NULL in the WinCreateWindow function to create a window that has no owner window. Some functions might return NULL, indicating that the given action applies to no window.

Window Size and Position

A window's size and position can be expressed as a bounding rectangle, given in coordinates relative to its parent. An application specifies the window's initial size and position when creating the window.

To use the system-default values for the initial size and position of a frame window, an application can specify the FCF_SHELLPOSITION frame-creation flag. The

application can change a window's size and position at any time. Figure 1-6 on page 1-15 indicates the size and position coordinates of a parent window and a child window.

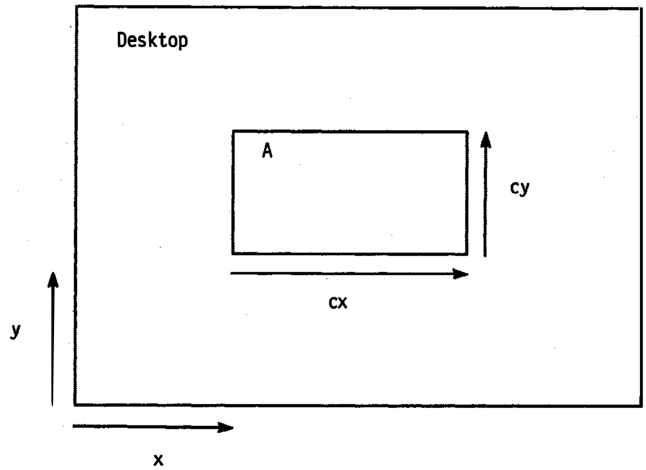


Figure 1-6. Window Sizing and Positioning

Notes:

1. The default coordinate system for a window specifies that the point (0,0) is at the lower-left corner of the window, with coordinates increasing as they go upward and to the right.
2. A window can be positioned anywhere in relation to its parent.

Size

A window's *size* (width and height) is given in pels, in the range 0 through 65535. A window can have 0 width and height; however, a window with 0 width or height is not drawn on the screen, even though it has the WS_VISIBLE style.

An application can create very large windows; however, it should check the size of the screen before enlarging a window size. One way to choose an appropriate size is to use the WinGetMaxPosition function to retrieve the size of the maximized window. A window that is larger than its maximized size will be larger than the screen also.

An application can retrieve the current size of the window by using the WinQueryWindowRect function.

Position

A window's *position* is defined as the *x,y* coordinates of its lower-left corner. These coordinates, sometimes called *window coordinates,* always are relative to the lower-left corner of the parent window. For example, a window having the coordinates (10,10) is placed 10 pels to the right of, and 10 pels up from, the lower-left corner of its parent window. Notice, however, that a window can be positioned anywhere in relation to its parent, but always relative to the parent's lower-left corner.

Adjusting a window's position can improve drawing performance. For example, an application could position a window so that its horizontal position is a multiple of 8, relative to the screen *origin* (the lower-left corner of the screen). Coordinates that

are multiples of 8 correspond to byte boundaries in the screen-memory bit map. It is usually faster to start drawing at a byte boundary.

By default, the system positions a frame window on a byte boundary; but an application can override this action by using the FCF_NOBYTEALIGN style when creating the window.

Size and Position Messages

A window receives messages when it changes size or position. Before a change is made, the system might send a WM_ADJUSTWINDOWPOS message to allow the window procedure to make final adjustments to the window's size and position. This message includes a pointer to an SWP structure that contains the requested width, height, and position. If the window procedure adjusts these values in the structure, the system uses the adjusted values to redraw the window. The WM_ADJUSTWINDOWPOS message is not sent if the change is a result of a call to the WinSetWindowPos function with the SWP_NOADJUST constant specified.

After a change has been made to a window, the system sends a WM_SIZE message to specify the new size of the window. If the window has the class style CS_MOVENOTIFY, the system also sends a WM_MOVE message, which includes the new position for the window. The system sends a WM_SHOW message if the visibility of the window has changed.

System Commands

An application that has a window with a system menu can change the size and position of that window by sending system commands. The system commands are generated when the user chooses commands from the system menu. An application can emulate the user action by sending a WM_SYSCOMMAND message to the window.

Following are some of the system commands:

Table 1-3. System Commands

Command	Description
SC_SIZE	Starts a Size command. The user can change the size of the window with a mouse and the keyboard.
SC_MOVE	Starts a Move command. The user can move the window with a mouse and the keyboard.
SC_MINIMIZE	Minimizes the window.
SC_MAXIMIZE	Maximizes the window.
SC_RESTORE	Restores a minimized or maximized window to its previous size and position.
SC_CLOSE	Closes the window. This command sends a WM_CLOSE message to the window. The window performs all tasks needed to clean up and destroy itself.

Window Data

Every window has an associated data structure. The window data structure contains all the information specified for the window at the time it was created and any additional information supplied for the window since that time. Although the exact size and meaning of the information in the window data structure are private to the system, an application can access any of the following data items via system-provided functions:

- Pointer to window-instance data structure
- Pointer to window procedure
- Parent-window handle
- Owner-window handle
- Handle of first child window
- Handle of next sibling window
- Window size and position (expressed as a rectangle)
- Window style
- Window identifier
- Update-region handle
- Message-queue handle.

An application can examine and modify this data by using functions such as WinQueryWindowUShort and WinSetWindowUShort. These functions let an application access data that is stored as 16-bit integers. Other functions let an application access data containing 32-bit integers and pointers. Several functions indirectly affect the data items in the window data structure. For example, the WinSubclassWindow function replaces the window-procedure pointer, and the WinSetWindowPos function changes the size and position of the window.

An application can extend the number of available data items in the window data structure by specifying a count of extra bytes when it registers the corresponding window class. Then, the window procedure can use these bytes to store information about the window. The WinQueryWindowUShort and WinSetWindowUShort functions give direct access to the extra bytes.

It generally is not a good idea to use direct storage in the window data. It is better to allocate a data structure dynamically and set a pointer to that data structure in the window words. This provides two advantages:

1. Most importantly, it is a symbolic way of referencing the data structure. It is very easy to make mistakes and provide the wrong offsets to WinQueryWindowUShort and so forth.

2. You now can add and remove fields without cross dependencies, because you now use *symbolic* references; whereas, when you use the technique of putting window words directly in the window data structure, you have to account for changed offsets.

Window Resources

Window resources are read-only data segments stored in an application's .EXE file or in a dynamic link library's .DLL file. Predefined PM window resources include keyboard accelerator tables, icons, menus, bit maps, dialog boxes, and so forth; these are not a regular part of the application window's code and data. Because, in most cases, window resources are not loaded into memory when the operating system runs a program, the resources can be shared by multiple instances of the same application.

Most window resources are stored in a format that is unique to each resource type. The application does not need to know these formats because the system translates them, as necessary, for use in PM functions. The following table lists the ten most commonly used PM window resource types.

Table 1-4. Presentation Manager-Defined Resource Types

Resource Identifier	Description
RT_ACCELTABLE	Keyboard accelerator table
RT_BITMAP	Bit map
RT_DIALOG	Dialog box template
RT_FONT	Font
RT_FONTDIR	Font directory
RT_MENU	Menu template
RT_MESSAGE	Message string
RT_POINTER	Icon or mouse
RT_RCDATA	Programmer-defined data
RT_STRING	Text string

To access these resources, you must prepare a *resource file* (ASCII file with the extension .RC). Then the ASCII resource file must be compiled into binary images using the resource compiler. The compiled resource file extension is .RES; it can be linked into your program's .EXE file or to a dynamic link library's .DLL file.

Maximized and Minimized Windows

A *maximized window* is a window that has been enlarged to fill the screen. Although a window's size can be set so that it fills the screen exactly, a maximized window is slightly different: the system automatically moves the window's title bar to the top of the screen and sets the WS_MAXIMIZED style for the window.

A *minimized window* is a window whose size has been reduced to exactly the size of an icon or, in the workplace shell, it disappears altogether (by default). Like a maximized window, a minimized window is more than just a window of a given size; typically, the system moves the (icon) minimized window to the lower part of the screen and sets the WS_MINIMIZED style for that window. The lower part of the screen is sometimes called the *icon area*. Unless the application specifies another position, the system moves a minimized window into the first available icon position in the icon area.

If a window is created with the WS_MAXIMIZED or WS_MINIMIZED styles, the system draws the window as a maximized or minimized window.

An application can restore maximized or minimized windows to their previous size and position by specifying the SWP_RESTORE flag in a call to the WinSetWindowPos function.

Window Visibility

A window that is a descendant of the desktop window can be either visible or invisible. The system displays a visible window on the screen. It hides an invisible window by not drawing it. If a window is visible, the user can supply input to the window and view the window's output. If a window is invisible, the window, in effect, is disabled. An invisible window can process messages from the system or from other windows, but it cannot process user input or display output. An application sets a window's visibility state when it creates the window. Later, a user or the application can change the visibility state.

The visible region of a window is the position clipped by any overlapping windows. These overlapping windows can be child windows or other main windows in the system. The visible region is defined by a set of one or more rectangles, as shown in Figure 1-7 on page 1-19.

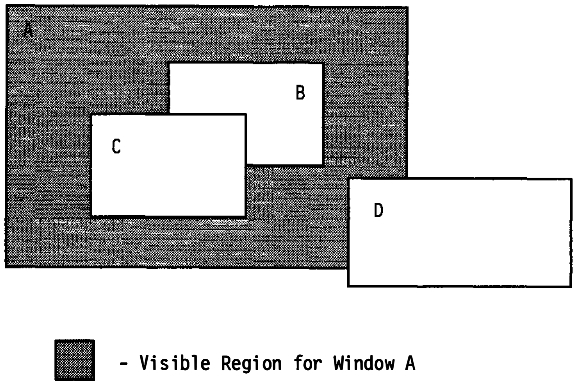


Figure 1-7. Visible Region for Window A

A window is visible if the WS_VISIBLE style is set for the window. By default, the WinCreateWindow function creates invisible windows unless the application specifies WS_VISIBLE. The application often hides a window to keep its operational details from the user. For example, an application can keep a new window invisible while it customizes the window's appearance. An application can determine whether a window has the WS_VISIBLE style by using the WinIsWindowVisible function.

Even if a window has the WS_VISIBLE style, the user might not be able to see the window on the screen because other windows completely overlap it, or it might have been moved beyond the edge of its parent. A visible window is subject to the clipping rules established by its parent-child relationship. If the window's parent window is not visible, the window will not be visible. Because a child window is drawn relative to its parent's lower-left corner, if the parent window is moved beyond the edge of the screen, the child window also will be moved. In other words, if a user moves the parent window containing the child window far enough off the edge of the screen, the user will not be able to see the child window, even though the child window and its parent window have the WS_VISIBLE style. To determine whether the user actually can see a window, an application can use the WinIsWindowShowing function.

Window Destruction

In general, an application must destroy all the windows it creates. It does this by using the WinDestroyWindow function. When a window is destroyed, the system hides the window, if it is visible, and then removes any internal data associated with the window. This invalidates the window handle so that it can no longer be used by the application.

An application destroys many of the windows it creates soon after creating them. For example, an application usually destroys a dialog window as soon as the application has sufficient input from the user to continue its task. An application eventually destroys the main window of the application (before terminating).

Destroying a window does not affect the window class from which the window was created. New windows still can be created using that class, and any existing windows of that class continue to operate.

When the application calls WinDestroyWindow, the system searches the descendancy tree for all windows below the specified window and destroys them from the bottom up, so each child receives WM_DESTROY before its parent. Each destroyed window is responsible for cleaning up its own resources in response to the WM_DESTROY message.

If a presentation space was created by the WinGetPS function for any of the windows to be destroyed, it must be released by calling the WinReleasePS function. The application must do this before calling the WinDestroyWindow function. If a presentation space is associated with the device context for the window, the application must disassociate or destroy the presentation space by using the GpiAssociate or GpiDestroyPS function before calling WinDestroyWindow. Failing to release a resource can cause an error.

For more information about presentation spaces and device contexts, see Chapter 28, "Painting and Drawing" on page 28-1.

If the window being destroyed is the active window, both the active and focus states are transferred to another window. The window that becomes the active window is the next window, as determined by the Alt+Esc key combination. The new active window then determines which window receives the keyboard focus.

Using Windows

The following sections explain how to create and use windows in an application, how to manage ownership and parent-child window relationships, and how to move and size windows.

Creating a Top-Level Frame Window

The main window in most applications is a top-level frame window. An application creates a top-level frame window by specifying the handle of the desktop window, or HWND_DESKTOP, as the hwndParent parameter in a call to the WinCreateStdWindow function.

Figure 1-8 on page 1-21 shows the main() function for a simple PM application. This function initializes the application, creates a message queue, and registers the window class for the client window before creating a top-level frame window.

```
#define IDR_RESOURCES 1

MRESULT EXPENTRY ClientWndProc(HWND, ULONG, MPARAM, MPARAM);

int main(VOID)
{
    HWND hwndFrame;
    HWND hwndClient;
    HMQ  hmq;
    QMSG qmsg;
    HAB  hab;

    /* Set the frame-window creation flags.                      */
    ULONG flFrameFlags =
        FCF_TITLEBAR      |   /* Title bar                        */
        FCF_SIZEBORDER    |   /* Size border                      */
        FCF_MINMAX        |   /* Minimize and maximize buttons.   */
        FCF_SYSMENU       |   /* System menu                      */
        FCF_SHELLPOSITION |   /* System-default size and position */
        FCF_TASKLIST ;        /* Add name to Task List.           */

    /* Initialize the application for PM                         */
    hab = WinInitialize(0);

    /* Create the application message queue.                     */
    hmq = WinCreateMsgQueue(hab, 0);

    /* Register the class for the client window.                 */
    WinRegisterClass(
        hab,                    /* Anchor block handle           */
        "MyPrivateClass",       /* Name of class being registered */
        (PFNWP)ClientWndProc,   /* Window procedure for class    */
        CS_SIZEREDRAW |         /* Class style                   */
        CS_HITTEST,             /* Class style                   */
        0);                     /* Extra bytes to reserve        */

    /* Create a top-level frame window with a client window      */
    /* that belongs to the window class "MyPrivateClass".        */
    hwndFrame = WinCreateStdWindow(
        HWND_DESKTOP,       /* Parent is desktop window.          */
        WS_VISIBLE,         /* Make frame window visible.         */
        &flFrameFlags,      /* Frame controls                     */
        "MyPrivateClass",   /* Window class for client            */
        NULL,               /* No window title                    */
        WS_VISIBLE,         /* Make client window visible .       */
        (HMODULE) 0,        /* Resources in application module    */
        IDR_RESOURCES,      /* Resource identifier                */
        NULL);              /* Pointer to client window handle    */

    /* Start the main message loop. Get messages from the        */
    /* queue and dispatch them to the appropriate windows.       */
    while (WinGetMsg(hab, &qmsg, 0, 0, 0))
          WinDispatchMsg(hab, &qmsg);

    /* Main loop has terminated. Destroy all windows and the     */
    /* message queue; then terminate the application.            */
    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);

    return 0;
}
```

Figure 1-8. Structure of a Simple Presentation Manager Application

Creating an Object Window

An application can create an object window by using the WinCreateWindow function and setting the desktop-object window as the parent window. The code fragment in Figure 1-9 shows how to create an object window.

```
#define ID_OBJWINDOW 2

HWND hwndObject;

hwndObject = WinCreateWindow(
    HWND_OBJECT,         /* Parent is object window.         */
    "MyObjClass",        /* Window class for client          */
    NULL,                /* Window text                      */
    0,                   /* No styles for object window      */
    0, 0,                /* Lower-left corner                */
    0, 0,                /* Width and height                 */
    NULL,                /* No owner                         */
    HWND_BOTTOM,         /* Inserts window at bottom of z-order */
    ID_OBJWINDOW,        /* Window identifier                */
    NULL,                /* No class-specific data           */
    NULL);               /* No presentation data             */
```

Figure 1-9. Creating an Object Window

Querying Window Data

An application can examine the values in the data structure associated with a window by using the WinQueryWindowUShort and WinQueryWindowULong functions. Each of these functions specifies a structure data item to examine. The index value can be an integer representing a zero-based byte index or a constant (QWS_) that identifies a specific item of data. The code fragment in Figure 1-10 obtains the programmer-defined identifier of the object window defined in the previous example:

```
HWND hwndObject;
USHORT usObjID;

usObjID = WinQueryWindowUShort(hwndObject, QWS_ID);
```

Figure 1-10. Getting the Window Identifier

Changing the Parent Window

An application can change a window's parent window by using the WinSetParent function. For example, in an application that uses child windows to display documents, you might want only the active document window to show a system menu. You can do this by changing that menu's parent window back and forth between the document window and the object window when WM_ACTIVATE messages are received. This technique is shown in the code fragment in Figure 1-11 on page 1-23:

```
switch (msg) {

case WM_ACTIVATE:  {

HWND hwndFrame, hwndSysMenu, hwnd;

    /* Get the handles of the frame window and system menu.        */
    hwndFrame = WinQueryWindow(hwnd, QW_PARENT);
    hwndSysMenu = WinWindowFromID(hwndFrame, FID_SYSMENU);

    /* If the window is being activated, make the frame window the */
    /* parent of the system menu. Otherwise, hide the system menu  */
    /* by making the object window the parent.                     */

    if ( SHORT1FROMMP(mp1))
        WinSetParent(hwndSysMenu, hwndFrame, TRUE);

    else
        WinSetParent(hwndSysMenu, HWND_OBJECT, TRUE);

    }

    return 0;
}
```

Figure 1-11. Changing the Parent Window

Finding a Parent, Child, or Owner Window

An application can determine the parent, child, and owner windows for any window by using the WinQueryWindow function. This function returns the window handle of the requested window.

The code fragment in Figure 1-12 determines the parent window of the given window:

```
HWND hwndParent;
HWND hwndMyWindow;

hwndParent = WinQueryWindow(hwndMyWindow, QW_PARENT);
```

Figure 1-12. Finding the Parent Window

The code fragment in Figure 1-13 determines the *topmost* child window (the child window in the top z-order position):

```
HWND hwndTopChild;
HWND hwndParent;

hwndTopChild = WinQueryWindow(hwndParent, QW_TOP);
```

Figure 1-13. Finding the Topmost Child Window

If a given window does not have an owner or child window, WinQueryWindow returns NULL.

Setting an Owner Window

An application can set the owner for a window by using the WinSetOwner function. Typically, after setting the owner, a window notifies the owner window of the new relationship by sending it a message.

The code fragment in Figure 1-14 shows how to set the owner window and send it a message:

```
#define NEW_OWNER 1

HWND hwndMyWindow;
HWND hwndNewOwner;

if (WinSetOwner(hwndMyWindow, hwndNewOwner))

    /* Send a notification message.                              */
    WinSendMsg(hwndNewOwner,   /* Sends to owner                  */
        WM_CONTROL,            /* Control message for notification */
        (MPARAM) NEW_OWNER,    /* Notification code               */
        NULL);                 /* No extra data                   */
```

Figure 1-14. Setting the Owner Window

A window can have only one owner, so WinSetOwner removes any previous owner.

Retrieving the Handle of a Child or Owned Window

A parent or owner window can retrieve the handle of a child or owned window by using the WinWindowFromID function and supplying the identifier of the child or owned window. WinWindowFromID searches all child and owned windows to locate the window with the given identifier. The window identifier is set when the application creates the child or owned window.

Typically, an owned window uses WinQueryWindow to get the handle of the owner window; then uses WinSendMsg to issue a notification message to its owner window.

The code fragment in Figure 1-15 retrieves the window handle of an owner window and sends the window a WM_ENABLE message:

```
HWND hwndOwned;
HWND hwndOwner;

case WM_CONTROL:
    switch (SHORT2FROMMP (mp2)) {
        case BN_CLICKED:
            hwndOwned = WinWindowFromID(hwndOwner,
            (ULONG)SHORT1FROMMP(mp1));
            WinSendMsg(hwndOwned, WM_ENABLE,
            (MPARAM)TRUE, (MPARAM) NULL);
            return 0;
        .
        . /* Check for other notification codes. */
        .
        }
```

Figure 1-15. Getting a Handle to an Owner or Child Window

An application also can retrieve the handle of a child window by using the WinWindowFromPoint function and supplying a point in the corresponding parent window.

Enumerating Top-Level Windows

An application can enumerate all top-level windows in the system by using the WinBeginEnumWindows and WinGetNextWindow functions. An application also can create a list of all child windows for a given parent window using WinBeginEnumWindows. This list contains the window handles of immediate child windows. By using WinGetNextWindow, the application then can retrieve the window handles, one at a time, from the list. When the application has finished using the list, it must release the list with the WinEndEnumWindows function.

The code fragment in Figure 1-16 shows how to enumerate all top-level windows (all immediate child windows of the desktop window):

```
HWND hwndTop;
HENUM henum;

/* Enumerate all top-level windows.          */
henum = WinBeginEnumWindows(HWND_DESKTOP);

/* Loop through all enumerated windows.      */
while (hwndTop = WinGetNextWindow(henum)) {
    .
    . /* Perform desired task on each window. */
    .
}

WinEndEnumWindows(henum);
```

Figure 1-16. Enumerating Top-Level Windows

Moving and Sizing a Window

An application can move a window by using the WinSetWindowPos function and specifying the SWP_MOVE constant. The function changes the position of the window to the specified position. The position is always given in coordinates relative to the parent window.

The code fragment in Figure 1-17 moves the window to the position (10,10):

```
HWND hwnd;

WinSetWindowPos(
    hwnd,                /* Window handle               */
    NULL,                /* Not used for moving and sizing */
    10, 10,              /* New position                */
    0, 0,                /* Not used for moving         */
    SWP_MOVE);           /* Move window                 */
```

Figure 1-17. Moving a Window

An application can set the size of a window by using the WinSetWindowPos function and specifying the SWP_SIZE constant. WinSetWindowPos changes the width and height of the window to the specified width and height.

An application can combine moving and sizing in a single function call, as shown in Figure 1-18.

```
HWND hwnd;

WinSetWindowPos(
    hwnd,                   /* Window handle                  */
    NULL,                   /* Not used for moving and sizing */
    10, 10,                 /* New position                   */
    200, 200,               /* Width and height               */
    SWP_MOVE | SWP_SIZE);   /* Move and size window.          */
```

Figure 1-18. Moving and Sizing a Window

An application can retrieve the current size and position of a window by using the WinQueryWindowPos function. This function copies the current information to an SWP structure.

The code fragment in Figure 1-19 uses the current size and position to change the height of the window, leaving the width and position unchanged:

```
HWND hwnd;
SWP swp;

WinQueryWindowPos(hwnd, &swp);
WinSetWindowPos(
    hwnd,                 /* Window handle                  */
    NULL,                 /* Not used for moving and sizing */
    0, 0,                 /* Not used for sizing            */
    swp.cx,               /* Current width                  */
    swp.cy + 200,         /* New height                     */
    SWP_SIZE);            /* Change the size.               */
```

Figure 1-19. Changing the Size of a Window

An application also can move and change the size of several windows at once by using the WinSetMultWindowPos function. This function takes an array of SWP structures. Each structure specifies the window to be moved or changed.

An application can move and size a window even if it is not visible, although the user is not able to see the effects of the moving and sizing until the window is visible.

Redrawing Windows

When the system moves a window or changes its size, it can invalidate all or part of that window. The system attempts to preserve the contents of the window and copy them to the new position; but if the window's size is increased, the window must fill the area exposed by the size change. If a window is moved from behind an overlapping window, any area formerly obscured by the other window must be drawn. In these cases, the system invalidates the exposed areas and sends a WM_PAINT message to the window.

An application can require that the system invalidate an entire window every time the window moves or changes size. To do this, the application sets the CS_SIZEREDRAW class style in the corresponding window class. Typically, this class style is selected for use in an application that uses a window's current size and position to determine how to draw the window. For example, a clock application always would draw the face of the clock so that it filled the window exactly.

An application also can explicitly specify which parts of the window to preserve during a move or size change. Before any change is made, the system sends a WM_CALCVALIDRECTS message to windows that do not have the style CS_SIZEREDRAW. This enables the window procedure to specify what part of the window to save and where to align it after the move or size change.

Changing the Z-Order of Windows

An application can move a window to the top or bottom of the z-order by passing the SWP_ZORDER constant to the WinSetWindowPos function. An application specifies where to move the window by specifying the HWND_TOP or HWND_BOTTOM constants.

The code fragment in Figure 1-20 uses WinSetWindowPos to change the z-order of a window:

```
HWND hwndParent;
HWND hwndNext;
HENUM henum;

    WinSetWindowPos(
        hwndNext,          /* Next window to move  */
        HWND_TOP,          /* Put window on top    */
        0, 0, 0, 0,        /* Not used for z-order */
        SWP_ZORDER);       /* Change z-order       */
```

Figure 1-20. Changing the Z-order of a Window

An application also can specify the window that the given window is to move behind. In this case, the application specifies the window handle instead of the HWND_TOP or HWND_BOTTOM constant.

```
HWND hwndParent;
HWND hwndNext;
HWND hwndExchange;
HENUM henum;

henum = WinBeginEnumWindows(hwndParent);

hwndExchange = WinGetNextWindow(henum);

/* hwndNext has top window; hwndExchange has window under the top. */
WinSetWindowPos(
    hwndNext,         /* Next window to move     */
    hwndExchange,     /* Put lower window on top */
    0, 0, 0, 0,       /* Not used for z-order    */
    SWP_ZORDER);      /* Change z-order          */

WinEndEnumWindows(henum);
```

Figure 1-21. Exchanging the Z-order of Windows

Showing or Hiding a Window

An application can show or hide a window by using the WinShowWindow function. This function changes the WS_VISIBLE style of a window to the specified setting. An application can also use the WinIsWindowVisible function to check the visibility of a window. This function returns TRUE if the window is visible.

Maximizing, Minimizing, and Restoring a Frame Window

An application can maximize, minimize, or restore a frame window by using the WinSetWindowPos function and specifying the constant SWP_MAXIMIZE, SWP_MINIMIZE, or SWP_RESTORE. Only a frame window can maximize and minimize by default. For any other window, an application must provide support for these actions in the corresponding window procedure.

Figure 1-22 shows how to maximize a frame window:

```
SWP swpCurrent;
HWND hwndFrame;

WinQueryWindowPos(hwndFrame, &swpCurrent);
WinSetWindowPos(
    hwndFrame,           /* Window handle             */
    NULL,                /* Not used to maximize      */
    swpCurrent.x,
    swpCurrent.y,        /* Stored for restoring window */
    swpCurrent.cx,
    swpCurrent.cy,       /* Stored for restoring window */
    SWP_MAXIMIZE | SWP_SIZE | SWP_MOVE);    /* Maximize */
```

Figure 1-22. Maximizing a Frame Window

Destroying a Window

An application can destroy a window by using the WinDestroyWindow function. Figure 1-23 shows how to create and then destroy a control window:

```
HWND hwndCtrl;
HWND hwndParent;

hwndCtrl = WinCreateWindow(hwndParent, WC_BUTTON, ...);

WinDestroyWindow(hwndCtrl);
```

Figure 1-23. Destroying a Window

Summary

Following are the OS/2 functions, messages, and data structures used with windows.

Table 1-5 (Page 1 of 3). Window Functions	
Window Creation Functions	
WinCreateWindow	The most direct way of creating a window. The window is of class ClassName and returns hwnd.
WinCreateStdWindow	Creates a main window. Requires an anchor block.
Window Destruction Functions	
WinDestroyWindow	Destroys a window and its child windows, and releases all their resources.
Window Data Functions	
WinQueryWindowUShort	Obtains the unsigned short integer value of a given window at a specified offset from the reserved window word's memory.
WinSetWindowUShort	Sets an unsigned, short integer value into the memory of the reserved window words.
WinQueryWindowULong	Obtains the unsigned long integer value of a given window, at a specified offset, from the memory of a reserved window word.
WinSetWindowULong	Sets an unsigned, long integer value into the memory of the reserved window words.
WinQueryWindowPtr	Retrieves a pointer value from the memory of the reserved window word.
WinSetWindowPtr	Sets a pointer value into the memory of the reserved window words.
WinSetWindowBits	Sets a number of bits into the memory of the reserved window words.
Window Relationship Functions	
WinSetParent	Sets the parent for hwnd to NewParent.

Table 1-5 (Page 2 of 3). Window Functions	
WinQueryWindow	Returns the handle of a window that has a specified relationship to a specified window.
WinSetOwner	Changes the owner of a specified window.
WinBeginEnumWindows	Begins the enumeration process for all the immediate child windows of a specified window.
WinGetNextWindow	Gets the window handle of the next window in a specified enumeration list.
WinEndEnumWindows	Ends the specified enumeration process.
WinIsChild	Tests to determine whether one window is a descendant of another.
WinQueryDesktopWindow	Returns the desktop window handle.
WinQueryObjectWindow	Returns the desktop-object window handle.
WinWindowFromID	Returns the handle of the child window with the specified ID.
WinWindowFromPoint	Finds the window, below a specified point, that is a descendant of a specified window.
WinMultWindowFromIDs	Finds the handles of child windows that belong to a specified window and that have window IDs within a specified range.
Window Size and Position Functions	
WinSetWindowPos	Facilitates the general positioning of a window.
WinQueryWindowPos	Obtains the size and position of a window.
WinSetMultWindowPos	An efficient means of repositioning multiple windows with one call, provided all windows being positioned have the same parent.
WinQueryWindowRect	Returns a window rectangle.
WinGetMinPosition	Returns the position to which a window is minimized.
Window Visibility Functions	
WinIsWindowShowing	Determines whether any part of the window, hwnd, is physically visible.
WinShowWindow	Sets the visibility state of a window.
WinIsWindowVisible	Returns the visibility state of a window.
Window Input Functions	
WinQueryActiveWindow	Returns the active window for HWND_DESKTOP or other parent window.
WinSetActiveWindow	Sets the main window as the active window.
WinQueryFocus	Returns the focus window; NULL if there is no focus window.
WinSetFocus	Sets the focus window.
WinQuerySysModalWindow	Returns the current system-modal window.
WinRequestMutexSem	Requests the ownership of a mutex semaphore or waits for a PM message.
WinSetSysModalWindow	Either sets a system-modal window or ends the system-modal state.

Table 1-5 (Page 3 of 3). Window Functions	
WinStartApp	Starts an application.
WinTerminate	Terminates an application thread's use of PM and releases all of its associated resources.
WinTerminateApp	Terminates an application started with WinStartApp.
WinWaitEventSem	Waits for an event semaphore to be posted or for a PM message.
WinWaitMuxWaitSem	Waits for a muxwait semaphore to clear or for a PM message.

Table 1-6. Window Messages

Message	Description
WM_ACTIVATE	Sent to a window as it gains or loses activation.
WM_ADJUSTWINDOWPOS	Sent to adjust a window's position. Not sent if SWP_NOADJUST is specified.
WM_CALCFRAMERECT	Occurs when an application uses the WinCalcFrameRect call.
WM_CALCVALIDRECTS	Sent from WinSetWindowPos and WinSetMultWindowPos to determine which areas of a window will be preserved if a window is sized and which should be redisplayed.
WM_CLOSE	Sent to a frame window to indicate that the window is being closed by the user.
WM_CREATE	Occurs when the application requests creation of a window.
WM_DESTROY	Occurs when the application requests destruction of a window.
WM_ENABLE	Sets the enable state of a window.
WM_MOVE	Occurs when a window with the style CS_MOVENOTIFY changes its absolute position.
WM_PAINT	Occurs when a window needs repainting.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the window parameters.
WM_SHOW	Occurs when a window's WS_VISIBLE state is being changed.
WM_SIZE	Occurs when a window changes its size.
WM_WINDOWPOSCHANGED	Sent to the window procedure of the window whose position is changed.

Table 1-7. Window Data Structures	
Data Structure	**Description**
CREATESTRUC	Create window.
WNDPARAMS	Window parameters.

Chapter 2. Messages and Message Queues

The OS/2 operating system uses messages and message queues to communicate with applications and the windows belonging to those applications. This chapter explains how to create and use messages and message queues in PM applications.

About Messages and Message Queues

Unlike traditional applications that take complete control of the computer's keyboard, mouse, and screen, PM applications must share these resources with other applications that are running at the same time. All applications run independently and rely on the operating system to help them manage shared resources. The operating system does this by controlling the operation of each application, communicating with each application when there is keyboard or mouse input or when an application must move and size its windows.

Messages

A *message* is information, a request for information, or a request for an action to be carried out by a window in an application.

The operating system, or an application, sends or posts a message to a window so that the window can use the information or respond to the request.

There are three types of messages:

- User-initiated
- Application-initiated
- System-initiated.

A user-initiated message is the direct result of a user action, such as selecting a menu item or pressing a key. An application-initiated message is generated by one window in the application to communicate with another window. System-initiated messages are generated by the interface as the indirect result of a user action (for example, resizing a window) or as the direct result of a system event (such as creating a window).

A message that requires an immediate response from a window is sent directly to the window by passing the message data as arguments to the window procedure. The window procedure carries out the request or lets the operating system carry out default processing for the message.

A message that does not require an immediate response from a window is *posted* (the message data is copied) to the application's *message queue*. The message queue is a storage area that the application creates to receive and hold its posted messages. Then, the application can retrieve a message at the appropriate time, sending it to the addressed window for processing.

Every message contains a *message identifier*, which is a 16-bit integer that indicates the purpose of the message. When a window processes a message, it uses the message identifier to determine what to do.

Every message contains a *window handle*, which identifies the window the message is for. The window handle is important because most message queues and window procedures serve more than one window. The window handle ensures that the application forwards the message to the proper window.

A message contains two *message parameters*—32-bit values that specify data or the location of data that a window uses when processing the message. The meaning and value of a message parameter depend on the message. A message parameter can contain an integer, packed bit flags, a pointer to a structure that contains additional data, and so forth. Some messages do not use message parameters and, typically, set the parameters to NULL. An application always checks the message identifier to determine how to interpret the message parameters.

A *queue message* is a QMSG data structure that contains six data items, representing the window handle, message identifier, two message parameters, message time, and mouse-pointer position. The time and position are included because most queue messages are input messages, representing keyboard or mouse input from the user. The time and position also help the application identify the context of the message. The operating system posts a queue message by filling the QMSG structure and copying it to a message queue.

A *window message* consists of the window handle, the message identifier, and two message parameters. A window message does not include the message time and mouse-pointer position, because most window messages are requests to perform a task that is not related to the current time or mouse-pointer position. The operating system sends a window message by passing these values, as individual arguments, to a window procedure.

Message Queues

Every PM application must have a *message queue*. A message queue is the only means an application has to receive input from the keyboard or mouse. *Only applications that create message queues can create windows.*

An application creates a message queue by using the WinCreateMsgQueue function. This function returns a handle that the application can use to access the message queue. After an application creates a message queue, the system posts messages intended for windows in the application to that queue. The application can retrieve queue messages by specifying the message-queue handle in the WinGetMsg function. It also can examine messages, without retrieving them, by using the WinPeekMsg function. When an application no longer needs the message queue, it can destroy the queue by using the WinDestroyMsgQueue function.

One message queue serves all the windows in a thread. This means a queue can hold messages for several windows. A message specifies the handle of the window to which it belongs so the application can forward a message easily to the appropriate window. The message loop recognizes a NULL window handle and the message is processed within the message loop rather than passed to WinDispatchMessage. See Figure 2-1 on page 2-4 for an example of an input-message processing loop.

An application that has more than one thread can create more than one message queue. The system allows one message queue for each thread. A message queue created by a thread belongs to that thread and has no connection to other queues in the application. When an application creates a window in a given thread, the system associates the window with the message queue in that thread. The system then posts all subsequent messages intended for that window to that queue.

Note: The recommended way to structure PM applications is to have at least two threads and two message queues. The first thread and message queue control all the user-interface windows, and the second thread and message queue control all the object windows.

Several windows can use one message queue; it is important that the message queue be large enough to hold all messages that possibly can be posted to it. An application can set the size of the message queue when it creates the queue by specifying the maximum number of messages the queue can hold. The default maximum number of messages is 10.

To minimize queue size, several types of posted messages are not actually stored in a message queue. Instead, the operating system keeps a record in the queue of the message being posted and combines any information contained in the message with information from previous messages. Timer, semaphore, and paint messages are handled this way. For example, if more than one WM_PAINT message is posted, the operating system combines the *update regions* for each into a single update region. Although there is no actual WM_PAINT message in the queue, the operating system constructs one WM_PAINT message with the single update region when an application uses the WinGetMsg function.

The operating system handles mouse and keyboard input messages differently from the way it handles other types of messages. The operating system receives all keyboard and mouse events, such as keystrokes and mouse movements, into the system message queue. The operating system converts these events into messages and posts them, one at a time, to the appropriate application message queue. The application retrieves the messages from its queue and dispatches them to the appropriate window, which processes the messages.

The operating system message queue usually is large enough to hold all input messages, even if the user types or moves the mouse very quickly. If the operating system message queue does run out of space, the system *ignores* the most recent keyboard input (usually by beeping to indicate the input is ignored) and collects mouse motions into a WM_MOUSEMOVE message.

Every message queue has a corresponding MQINFO data structure that specifies the identifiers of the process and thread that own the message queue and gives a count of the maximum number of messages the queue can receive. An application can retrieve the structure by using the WinQueryQueueInfo function.

A message queue also has a current status that indicates the types of messages currently in the queue. An application can retrieve the queue status by using the WinQueryQueueStatus function. An application also can use the WinPeekMsg function to examine the contents of a message queue. WinPeekMsg checks for a specific message or range of messages in the queue and gives the application the option of removing messages from the queue. An application *can* call the WinQueryQueueStatus function to determine the contents of the queue before calling the WinPeekMsg or WinGetMsg function to remove a message from the queue.

Message Handling

To handle and process messages, an application can use a *message loop* and the *window procedure*. These terms are explained in the following two sections.

Message Loops

Every application with a message queue is responsible for retrieving the messages from that queue. An application can do this by using a message loop, usually in the application's main function, that retrieves messages from the message queue and dispatches them to the appropriate windows. The message loop consists of two

calls: one to the WinGetMsg function; the other to the WinDispatchMsg function. The message loop has the following form:

```
HAB hab;
QMSG qmsg;

while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
    WinDispatchMsg(hab, &qmsg);
```

An application starts the message loop after creating the message queue and at least one application window. Once started, the message loop continues to retrieve messages from the message queue and to dispatch (send) them to the appropriate windows. WinDispatchMsg sends each message to the window specified by the window handle in the message.

Figure 2-1 illustrates the typical routing of an input message through the operating system's and application's message loops.

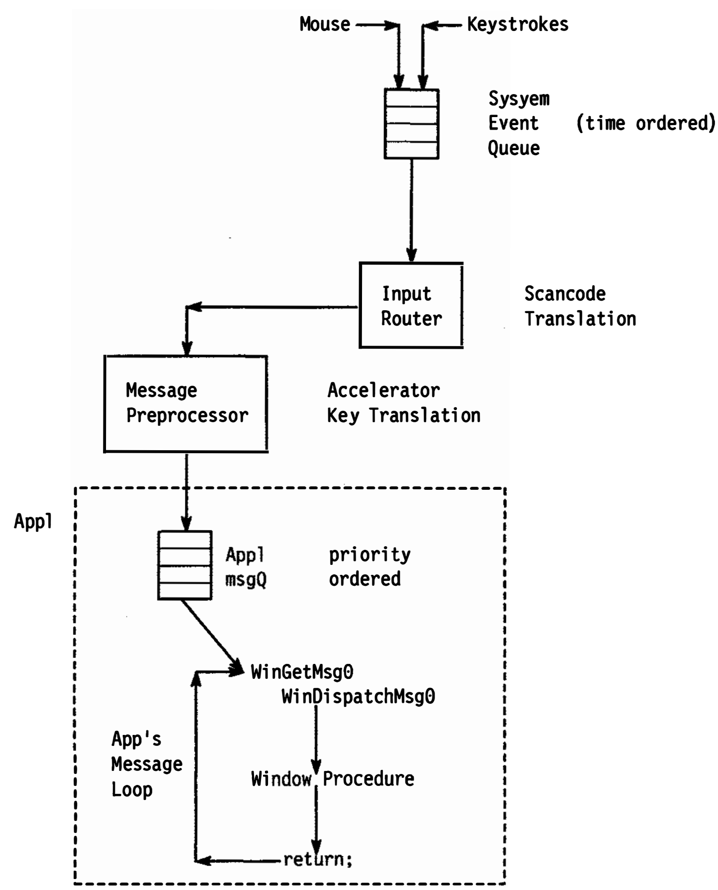


Figure 2-1. Input Message Processing Loop

Only one message loop is needed for a message queue, even if the queue contains messages for more than one window. Each queue message is a QMSG structure that contains the handle of the window to which the message belongs. WinDispatchMsg always dispatches the message to the proper window. WinGetMsg retrieves messages from the queue in first-in, first-out (FIFO) order, so the messages are dispatched to windows in the same order they are received.

If there are no messages in the queue, the operating system temporarily stops processing the WinGetMsg function until a message arrives. This means that CPU time that, otherwise, would be spent waiting for a message can be given to the applications (or threads) that do have messages in their queues.

The message loop continues to retrieve and dispatch messages until WinGetMsg retrieves a WM_QUIT message. This message causes the function to return FALSE, terminating the loop. In most cases, terminating the message loop is the first step in terminating the application. An application can terminate its own loop by posting the WM_QUIT message in its own queue.

An application can modify its message loop in a variety of ways. For example, it can retrieve messages from the queue without dispatching them to a window. This is useful for applications that post messages without specifying a window. (These messages apply to the application rather than a specific window; they have NULL window handles.) Also, an application can direct the WinGetMsg function to search for specific messages, leaving other messages in the queue. This is useful for applications that temporarily need to bypass the usual FIFO order of the message queue.

Window Procedures

A *window procedure* is a function that receives and processes all input and requests for action sent to the windows. Every window class has a window procedure; every window created using that class uses that window procedure to respond to messages.

The system sends a message to the window procedure by passing the message data as arguments. The window procedure takes the appropriate action for the given message. Most window procedures check the message identifier, then use the information specified by the message parameters to carry out the request. When it has completed processing the message, the window procedure returns a message result. Each message has a particular set of possible return values. The window procedure must return the appropriate value for the processing it performed.

A window procedure cannot ignore a message. If it does not process a message, it must pass the message back to the operating system for default processing. The window procedure does this by calling the WinDefWindowProc function to carry out a default action and return the message result. Then, the window procedure must return this value as its own message result.

A window procedure commonly processes messages for several windows. It uses the window handle specified in the message to identify the appropriate window. Most window procedures process just a few types of messages and pass the others on to the operating system by calling WinDefWindowProc.

Posting and Sending Messages

Any application can post and send messages. Like the operating system, an application *posts* a message by copying it to a message queue. It *sends* a message by passing the message data as arguments to a window procedure. To post and send messages, an application uses the WinPostMsg and WinSendMsg functions.

An application posts a message to notify a specific window to perform a task. The WinPostMsg function creates a QMSG structure for the message and copies the message to the message queue corresponding to the given window. The application's message loop eventually retrieves the message and dispatches it to

the appropriate window procedure. For example, one message commonly posted is WM_QUIT. This message terminates the application by terminating the message loop.

An application sends a message to cause a specific window procedure to carry out a task immediately. The WinSendMsg function passes the message to the window procedure corresponding to the given window. The function waits until the window procedure completes processing and then returns the message result. Parent and child windows often communicate by sending messages to each other. For example, a parent window that has an entry-field control as its child window can set the text of the control by sending a message to the child window. The control can notify the parent window of changes to the text (carried out by the user) by sending messages back to the parent window.

Occasionally, an application might need to send or post a message to all windows in the system. For example, if the application changes a system value, it must notify all windows about the change by sending a WM_SYSVALUECHANGED message. An application can send or post messages to any number of windows by using the WinBroadcastMsg function. The options in WinBroadcastMsg determine whether the message is sent or posted and specify the windows that will receive the message.

Any thread in the application can post a message to a message queue, even if the thread has no message queue of its own. However, only a thread that has a message queue can send a message. Sending a message between threads is relatively uncommon. For one reason, sending a message is costly in terms of system performance. If an application posts a message between threads, it is likely to be a semaphore message, which permits window procedures to manage a shared resource jointly.

An application can post a message without specifying a window. If the application supplies a NULL window handle when it calls the WinPostMsg function, the function posts the message to the queue associated with the current thread. The application must process the message in the message loop. This is one way to create a message that applies to the entire application instead of to a specific window.

A window procedure can determine whether it is processing a message sent by another thread by using the WinInSendMsg function. This is useful when message processing depends on the origin of the message.

A common programming error is to assume that the WinPostMsg function always succeeds. It fails when the message queue is full. An application should check the return value of the WinPostMsg function to see whether the message was posted. In general, if an application intends to post many messages to the queue, it should set the message queue to an appropriate size when it creates the queue. The default message-queue size is 10 messages.

Message Types

This section describes the three types of OS/2 messages:

- System-defined
- Application-defined
- Semaphore.

System-Defined Messages

There are many *system-defined* messages that are used to control the operations of applications and to provide input and other information for applications to process. The system sends or posts a system-defined message when it communicates with an application. An application also can send or post system-defined messages. Usually, applications use these messages to control the operation of control windows created by using preregistered window classes.

Each system message has a unique message identifier and a corresponding symbolic constant. The symbolic constant, defined in the system header files, states the purpose of the message. For example, the WM_PAINT constant represents the paint message, which requests that a window paint its contents.

The symbolic constants also specify the *message category*. System-defined messages can belong to several categories; the prefix identifies the type of window that can interpret and process the messages. The following table lists the prefixes and their related message categories:

Table 2-1. Message Categories

Prefix	Message category
BKM_	Notebook control
BM_	Button control
CBM_	Combination-box control
CM_	Container control
EM_	Entry-field control
LM_	List-box control
MLM_	Multiple-line entry field control
MM_	Menu control
SBM_	Scroll-bar control
SLM_	Slider control
SM_	Static control
TBM_	Title-bar control
VM_	Value set control
WM_	General window

General window messages cover a wide range of information and requests, including:

- Mouse and keyboard-input
- Menu- and dialog-input
- Window creation and management
- Dynamic data exchange (DDE).

Application-Defined Messages

An application can create messages to use in its own windows. If an application does create messages, the window procedure that receives the messages must interpret them and provide the appropriate processing.

The operating system reserves the message-identifier values in the range *0x0000* through *0x0FFF* (the value of WM_USER - 1) for system-defined messages. Applications cannot use these values for their private messages.

Values in the range *0x1000* (the value of WM_USER) through *0xBFFF*, however, are available for message identifiers, defined by an application, for use in that application.

Warning: It is very important that applications do not broadcast messages in the *0x1000* through *0xBFFF* range because of the risk of misinterpretation by other applications.

Values in the range *0xC000* through *0xFFFF* are reserved for message identifiers that an application defines and registers with the system atom table; these can be used in any application. Values above *0xFFFF (0x00010000* through *0xFFFFFFFF)* are reserved for future use; applications must not use messages in this range.

Semaphore Messages

A *semaphore message* provides a way of signaling, through the message queue, the end of an event. An application uses a semaphore message the same way it uses system semaphore functions—to coordinate events by passing signals. A semaphore message often is used in conjunction with system semaphores.

There are four semaphore messages:

WM_SEM1
WM_SEM2
WM_SEM3
WM_SEM4.

An application posts one of these messages to signal the end of a given event. The window that is waiting for the given event receives the semaphore message when the message loop retrieves and dispatches the message.

Each semaphore message includes a bit flag that an application can use to uniquely identify the 32 possible semaphores for each semaphore message. The application passes the bit flag (with the appropriate bit set) as a message parameter with the message. The window procedure that receives the message then uses the bit flag to identify the semaphore.

To save space, the system does not store semaphore messages in the message queue. Instead, it sets a record in the queue, indicating that the semaphore message has been received, and then combines the bit flag for the message with the bit flags from previous messages. When the window procedure eventually receives the message, the bit flag specifies each semaphore message posted since the last message was retrieved.

Message Priorities

The WinGetMsg function retrieves messages from the message queue based on message priority. WinGetMsg retrieves messages with higher priority first. If it finds more than one message at a particular priority level, it retrieves the oldest message first. Messages have the following priorities:

Table 2-2. Message Priorities	
Priority	**Message**
1	WM_SEM1
2	Messages posted using WinPostMsg
3	Input messages from the keyboard or mouse
4	WM_SEM2
5	WM_PAINT
6	WM_SEM3
7	WM_TIMER
8	WM_SEM4

Message Filtering

An application can choose specific messages to retrieve from the message queue (and ignore other messages) by specifying a message filter with the WinGetMsg or WinPeekMsg functions. The message filter is a range of message identifiers (specified by a first and last identifier), a window handle, or both. The WinGetMsg and WinPeekMsg functions use the *message filter* to select the messages to retrieve from the queue. Message filtering is useful if an application needs to search ahead in the message queue for messages that have a lower priority or that arrived in the queue later than other less important messages.

Any application that filters messages must ensure that a message satisfying the message filter can be posted. For example, filtering for a WM_CHAR message in a window that does not receive keyboard input prevents the WinGetMsg function from returning. Some messages, such as WM_COMMAND, are generated from other messages; filtering for them also can prevent WinGetMsg from returning.

To filter for mouse, button, and DDE messages, an application can use the following constants:

WM_MOUSEFIRST and WM_MOUSELAST
WM_BUTTONCLICKFIRST and WM_BUTTONCLICKLAST
WM_DDE_FIRST and WM_DDE_LAST.

Using Messages

This section explains how to perform the following tasks:

- Create a message queue and message loop.
- Examine the message queue.
- Post and send messages between windows.
- Broadcast a message to multiple windows.
- Use message macros.

Creating a Message Queue and Message Loop

An application needs a message queue and message loop to process messages for its windows. An application creates a message queue by using the WinCreateMsgQueue function. An application creates a message loop by using the WinGetMsg and WinDispatchMsg functions. The application must create and show at least one window after creating the queue but before starting the message loop.

The following code fragment shows how to create a message queue and message loop:

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2);

HAB hab;

int main(VOID)
{
    HMQ hmq;
    QMSG qmsg;
    HWND hwndFrame, hwndClient;
    ULONG flFrameFlags = FCF_TITLEBAR      | FCF_SYSMENU |
                         FCF_SIZEBORDER    | FCF_MINMAX  |
                         FCF_SHELLPOSITION | FCF_TASKLIST;

    /* Initialize the application for Presentation Manager interface.      */
    hab = WinInitialize(0);

    /* Create the application message queue.                                */
    hmq = WinCreateMsgQueue(hab, 0);

    /* Register the window class for your client window.                    */
    WinRegisterClass(hab,                 /* Anchor block handle            */
                     "MyClientClass",     /* Class name                     */
                     (PFNWP) ClientWndProc,/* Window procedure              */
                     CS_SIZEREDRAW,       /* Class style                    */
                     0);                  /* Extra bytes to reserve         */

    /* Create a main window.                                                */
    hwndFrame = WinCreateStdWindow(
                    HWND_DESKTOP,         /* Parent window handle           */
                    WS_VISIBLE,           /* Style of frame window          */
                    &flFrameFlags,        /* Frame controls                 */
                    "MyClientClass",      /* Window class for client        */
                    (PSZ) NULL,           /* No title-bar text              */
                    WS_VISIBLE,           /* Style of client window         */
                    (HMODULE) NULL,       /* Module handle for resources */
                    0,                    /* No resource identifier         */
                    &hwndClient);         /* Pointer to client handle       */

    /* Start the message loop.            */
    while (WinGetMsg(hab, &qmsg, (HWND) NULL, 0, 0))
        WinDispatchMsg(hab, &qmsg);

    /*. Destroy the main window.          */
    WinDestroyWindow(hwndFrame);

    /* Destroy the message queue.         */
    WinDestroyMsgQueue(hmq);

    /* Terminate the application.         */
    WinTerminate(hab);
}
```

Both the WinGetMsg and WinDispatchMsg functions take a pointer to a QMSG structure as a parameter. If a message is available, WinGetMsg copies it to the QMSG structure; WinDispatchMsg then uses the data in the structure as arguments for the window procedure.

Occasionally, an application might need to process a message before dispatching it. For example, if a message is posted but the destination window is not specified (that is, the message contains a NULL window handle), the application must process the message to determine which window should receive the message. Then the WinDispatchMsg function can forward the message to the proper window. The following code fragment shows how the message loop can process messages that have NULL window handles:

```
HAB hab;
QMSG qmsg;

while (WinGetMsg (hab, &qmsg, (HWND) NULL, 0, 0)) {
    if (qmsg.hwnd == NULL) {
        .
        . /* Process the message. */
        .
    }
    else
        WinDispatchMsg (hab, &qmsg);
    }
```

Examining the Message Queue

An application can examine the contents of the message queue by using the WinPeekMsg or WinQueryQueueStatus function. It is useful to examine the queue if the application starts a lengthy operation that additional user input might affect, or if the application needs to look ahead in the queue to anticipate a response to user input.

An application can use WinPeekMsg to check for specific messages in the message queue. This function is useful for extracting messages for a specific window from the queue. It returns immediately if there is no message in the queue. An application can use WinPeekMsg in a loop without requiring the loop to wait for a message to arrive. The following code fragment checks the queue for WM_CHAR messages:

```
HAB hab;
QMSG qmsg;

if (WinPeekMsg(hab, &qmsg, (HWND) NULL, WM_CHAR, WM_CHAR, PM_NOREMOVE)){
          .
          . /* Process the message. */
          .
}
```

An application also can use the WinQueryQueueStatus function to check for messages in the queue. This function is very fast and returns information about the kinds of messages available in the queue and which messages have been posted recently. Most applications use this function in message loops that need to be as fast as possible.

Posting a Message to a Window

An application can use the WinPostMsg function to post a message to a window. The message goes to the window's message queue. The following code fragment posts the WM_QUIT message:

```
HWND hwnd;

if (!WinPostMsg(hwnd, WM_QUIT, NULL, NULL)){

    /* Message was not posted. */
}
```

The WinPostMsg function returns FALSE if the queue is full, and the message cannot be posted.

Sending a Message to a Window

An application can use the WinSendMsg function to send a message directly to a window. An application uses this function to send messages to child windows. For example, the following code fragment sends an LM_INSERTITEM message to direct a list-box control to add an item to the end of its list:

```
HWND hwndListBox;
static CHAR szWeekday[] = "Tuesday";

WinSendMsg(hwndListBox,
           LM_INSERTITEM,
           (MPARAM)LIT_END,
           MPFROMP(szWeekday));
```

WinSendMsg calls the window's window procedure and waits for it to handle the message and return a result. An application can send a message to any window in the system, as long as the application has the handle of the target window. The message queue does not store the message; however, the thread making the call must have a message queue.

Broadcasting a Message

An application can send a message to multiple windows by using the WinBroadcastMsg function. Often this function is used to broadcast the WM_SYSVALUECHANGED message after an application changes a system value. The following code fragment shows how to broadcast this message to all frame windows in all applications:

```
HWND hwnd;

WinBroadcastMsg(
    hwnd,                                  /* Window handle        */
    WM_SYSVALUECHANGED,                    /* Message identifier   */
    NULL,                                  /* No message parameters */
    NULL,
    BMSG_FRAMEONLY | BMSG_POSTQUEUE);      /* All frame windows     */
```

An application can broadcast messages to all windows, just frame windows, or just the windows in the application.

Using Message Macros

The system header files define several macros that help create and interpret message parameters.

One set of macros helps you construct message parameters. These macros are useful for sending and posting messages. For example, the following code fragment uses the MPFROMSHORT macro to convert a 16-bit integer into the 32-bit message parameter:

```
HWND hwndButton;

WinSendMsg(hwndButton, BM_SETCHECK, MPFROMSHORT(1), NULL);
```

A second set of macros helps you extract values from a message parameter. These macros are useful for handling messages in a window procedure. The following code fragment determines whether the window receiving the WM_FOCUSCHANGE message is gaining or losing the keyboard focus. The fragment uses the SHORT1FROMMP macro to extract the focus-change flag, the SHORT2FROMMP macro to extract the focus flag, and the HWNDFROMMP macro to extract the window handle.

```
USHORT fsFocusChange;
MPARAM mp1, mp2;
HWND hwndGainFocus;


case WM_FOCUSCHANGE:
    fsFocusChange = SHORT2FROMMP(mp2);    /* Gets focus-change flags  */
    if (SHORT1FROMMP(mp2))                /* Gaining or losing focus? */
        hwndGainFocus = HWNDFROMMP(mp1);
```

A third set of macros helps you construct a message result. These macros are useful for returning message results in a window procedure, as the following code fragment illustrates:

```
return (MRFROM2SHORT(1, 2));
```

Summary

Following are the functions and structures used with OS/2 messages and message queues.

Table 2-3. Commonly Used Message and Message Queue Functions

Function Name	Description
WinCreateMsgQueue	Creates a message queue.
WinDefDlgProc	Invokes the default dialog procedure.
WinDefWindowProc	Invokes the default window procedure.
WinDestroyMsgQueue	Destroys the message queue.
WinDispatchMsg	Invokes a window procedure.
WinGetMsg	Gets a message from the thread's message queue and returns **msg** when a message conforming to the filtering criteria is available.
WinPeekMsg	Inspects the thread's message queue and returns to the application with or without a message.
WinPostMsg	Posts a message to the message queue associated with the window defined by **hwnd**.
WinSendDlgItemMsg	Sends a message to the dialog item defined by **Item** in the dialog window specified by **Dlg**
WinSendMsg	Sends a message with identity **Msgid** to **hwnd**.

Table 2-4. Seldom-Used Message and Message Queue Functions

Function Name	Description
WinBroadcastMsg	Broadcasts a message to multiple windows.
WinCallMsgFilter	Calls a message-filter hook.
WinInSendMsg	Determines whether the current thread is processing a message sent by another thread.
WinPostQueueMsg	Posts a message to a message queue.

Table 2-5 (Page 1 of 2). Almost-Never Used Message and Message Queue Functions

Function Name	Description
WinQueryMsgPos	Returns the pointer position, in screen coordinates, when the last message obtained from the current message queue is posted.
WinQueryQueueInfo	Returns the information for the specified queue.
WinQueryQueueStatus	Returns a code indicating the status of the message queue associated with the caller.
WinRegisterUserMsg	Registers a user message and defines its parameters.
WinSetClassMsgInterest	Sets the message interest of a message class.
WinSetMsgInterest	Sets a window's message interest.
WinSetMsgMode	Indicates the mode for the generation and processing of messages for the private window class of an application.

Table 2-5 (Page 2 of 2). Almost-Never Used Message and Message Queue Functions	
Function Name	**Description**
WinTranslateAccel	Translates a WM_CHAR message.
WinWaitMsg	Waits for a filtered message.

Table 2-6. Message and Message Queue Structures	
Structure Name	**Description**
HMQ	Message-queue handle.
MQINFO	Message-queue information structure.
QMSG	Message structure.

Chapter 3. Window Classes

A *window class* determines which styles and which window procedure are given to a window when it is created. This chapter explains how a PM application creates and uses window classes.

About Window Classes

Every window is a member of a window class. An application must specify a window class when it creates a window. Each window class has an associated *window procedure* that is used by all windows of the same class. The window procedure handles messages for all windows of that class and, therefore, controls the behavior and appearance of the window.

A window class must be *registered* before an application can create a window of that class. Registering a window class associates a window procedure and class styles with a class name. When an application specifies the class name in a window-creation function such as WinCreateWindow, the system creates a window that uses the window procedure and styles associated with the class name.

An application can register private classes or use preregistered public window classes.

Private Window Classes

A *private window class* is any class registered within an application. An application registers a private class by calling the WinRegisterClass function. A private class cannot be shared with other applications. When an application terminates, the system removes any data associated with the application's private window classes.

An application can register a private class anytime but, typically, does so as part of application initialization. To register a private class during application initialization, the application also must call WinInitialize and, usually, WinCreateMsgQueue before class registration.

An application cannot de-register a private window class; it remains registered and available until the application terminates.

When an application registers a private window class, it must supply the following information:

- Class name
- Class styles
- Window procedure
- Window data size.

Class Name

The *class name* identifies the window class. The application uses this name in the window-creation functions to specify the class of the window being created. The class name can be a character string or an atom, and it must be unique within the application. The system checks as to whether a public class or a class already registered by the application has the same name. If the class name is not unique to that application, the system returns an error.

Class Styles

Each window class has one or more values, called *class styles*, that tell the system which initial window styles to give a window created with that class. An application sets the class styles for a private window class when it registers the class. Once a class is registered, the application cannot change the styles.

An application can specify one or more of the following class styles in the WinRegisterClass function, combining them as necessary by using the bitwise OR operator:

Table 3-1. Class Styles

Style Name	Description
CS_CLIPCHILDREN	Prevents a window from painting over its child windows, but increases the time necessary to calculate the visible region. This style usually is not necessary, because if the parent and child windows overlap and are both invalidated, the operating system draws the parent window before drawing the child window. If the child window is invalidated independently of the parent window, the system redraws only the child window. If the update region of the parent window does not intersect the child window, drawing the parent window causes the child window to be redrawn. This style is useful to prevent a child window containing a complex graphic from being redrawn unnecessarily.
CS_CLIPSIBLINGS	Prevents a window from painting over its sibling windows. This style protects sibling windows but increases the time necessary to calculate the visible region. This style is appropriate for windows that overlap and have the same parent window.
CS_FRAME	Identifies the window as a frame window.
CS_HITTEST	Directs the operating system to send WM_HITTEST messages to the window whenever the mouse pointer moves in the window.
CS_MOVENOTIFY	Directs the system to send WM_MOVE messages to the window whenever the user moves the window.
CS_PARENTCLIP	Extends a window's visible region to include that of its parent window. This style simplifies the calculation of the child window's visible region but, potentially, is dangerous, because the parent window's visible region is usually larger than the child window.
CS_SAVEBITS	Saves the screen area under a window as a bit map. When the user hides or moves the window, the system restores the image by copying the bits; there is no need to add the area to the uncovered window's update region. This style can improve system performance, but also can consume a great deal of memory. It is recommended only for transient windows such as menus and dialog windows—*not* for main application windows.
CS_SIZEREDRAW	Causes the window to receive a WM_PAINT message and be completely invalidated whenever the window is resized, even if it is made smaller. (Typically, only the uncovered area of a window is invalidated when a window is resized.) This class style is useful when an application scales graphics to fill the window.
CS_SYNCPAINT	Causes the window to receive WM_PAINT messages immediately after a part of the window becomes invalid. Without this style, the window receives WM_PAINT messages only if no other message is waiting to be processed.

Window Procedure

The window procedure for a window class processes all messages sent or posted to all windows of that class. It is the chief component of the window class because it controls the appearance and behavior of each window created with the class. Window procedures are shared by all windows of a class, so an application must ensure that no conflicts arise when two windows of the same class attempt to access the same global data. In other words, the window procedure must protect global data and other shared resources.

Window Data Size

The system creates a window data structure for each window, which includes extra space that an application can use to store additional data about a window. An application specifies the number of extra bytes to allocate in the WinRegisterClass function. All windows of the same class have the same amount of window data space.

An application can store window data in a window's data structure by using the WinSetWindowUShort and WinSetWindowULong functions. It can retrieve data by using the WinQueryWindowUShort and WinQueryWindowULong functions.

Custom Window Styles

An application that registers a window class also can support its own set of styles for windows of that class. Standard window styles—for example, WS_VISIBLE and WS_SYNCPAINT—still apply to these windows. A window style is a 32-bit integer, and only the high 16 bits are used for the standard window styles; an application can use the low 16 bits for custom styles specific to a window class.

The operating system has unique window styles for all preregistered window classes. Styles such as FS_BORDER and BS_PUSHBUTTON are processed by the window procedure for the corresponding class. This means that an application can build the support for its own window styles into the window procedure for its private class. A window style designed for one window class will not work with another window class.

Public Window Classes

Public window classes are registered during system initialization. Their window procedures are in dynamic link libraries. Therefore, to use a public window class, an application need not register it. Nor does the application need to import the window procedure for a public window class because the system resolves references to the window procedure.

An application cannot use a public window class name when it registers a private window class.

System-Defined Public Window Classes

The system provides a number of public window classes that support menus, frame windows, control windows, and dialog windows. An application can create a window of a system-defined public window class by specifying one of the following class name constants in a call to WinCreateWindow:

Table 3-2. Public Window Classes	
Class Name	**Description**
WC_BUTTON	Consists of buttons and boxes the user can select by clicking the pointing device or using the keyboard.
WC_COMBOBOX	Creates a combination-box control, which combines a list-box control and an entry-field control. It enables the user to enter data either by typing in the entry field or by choosing from the list in the list box.
WC_CONTAINER	Creates a control in which the user can group objects in a logical manner. A container can display those objects in various formats or views. The container control supports drag and drop so the user can place information in a container by simply dragging and dropping.
WC_ENTRYFIELD	Consists of a single line of text that the user can edit.
WC_FRAME	A composite window class that can contain child windows of many of the other window classes.
WC_LISTBOX	Presents a list of text items from which the user can make selections.
WC_MENU	Presents a list of items that can be displayed horizontally as menu bars, or vertically as pull-down menus. Usually menus are used to provide a command interface to applications.
WC_NOTEBOOK	Creates a control for the user that is displayed as a number of pages. The top page is visible, and the others are hidden, with their presence being indicated by a visible edge on each of the back pages.
WC_SCROLLBAR	Consists of window scroll bars that let the user scroll the contents of the associated window.
WC_SLIDER	Creates a control that is usable for producing approximate (analog) values or properties. Scroll bars were used for this function in the past, but the slider provides a more flexible method of achieving the same result, with less programming effort.
WC_SPINBUTTON	Creates a control that presents itself to the user as a scrollable ring of choices, giving the user quick access to the data. The user is presented only one item at a time, so the spin button should be used with data that is intuitively related.
WC_STATIC	Simple display items that do not respond to keyboard or pointing device events.
WC_TITLEBAR	Displays the window title or caption and lets the user move the window's owner.
WC_VALUESET	Creates a control similar in function to radio buttons but provides additional flexibility to display graphical, textual, and numeric formats. The values set with this control are mutually exclusive.

Each system-defined public window class has a corresponding set of window styles that an application can use to customize a window of that class. For example, a window created with the WC_BUTTON class has styles that include BS_PUSHBUTTON and BS_CHECKBOX. Window styles enable you to customize aspects of a window's behavior and appearance. The application specifies the window styles in the WinCreateWindow function.

Custom Public Window Classes

An application can create a custom public window class, but it must do so during system initialization. Only the shell can register a public window class, and it can do so only when the system starts. Registering a public window class requires a special load entry in the os2.ini file. That entry instructs the shell to load a dynamic link library whose initialization routine registers the window class. Custom public window classes must be registered using WinRegisterClass and must have the class style CS_PUBLIC. If a custom public window class registered this way has the same name as an existing public window class, the custom class replaces the original class.

If a dynamic link library replaces an existing public window class, the library can save the address of the original window procedure and use the address to subclass the original window class. The dynamic link library retrieves the original window procedure address using the WinQueryClassInfo function. The custom window procedure then passes unprocessed messages to the original window procedure instead of calling WinDefWindowProc.

When subclassing a public window class, the custom public window procedure must not make the window data size smaller than the original window data size, because all public window classes that the operating system defines use 4 extra bytes for storing a pointer to custom window data. This size is guaranteed only for public window classes defined by the operating system dynamic link libraries.

Class Data

An application can examine public window class data by using the WinQueryClassInfo and WinQueryClassName functions. An application retrieves the name of the class for a given window by using the WinQueryClassName function. If the window is one of the preregistered public window classes, the name returned is in the form *#nnnnn*, where *nnnnn* is up to 5 digits, representing the value of the window class constant. Using this window class name, the application can call WinQueryClassInfo to retrieve the window class data. WinQueryClassInfo copies the class style, window procedure address, and window data size to a CLASSINFO data structure.

Using Window Classes

This section explains how to perform the following tasks:

- Register a private window class.
- Register an imported window procedure.

Registering a Private Window Class

An application can register a private window class at any time by using the WinRegisterClass function. You must define the window procedure in the application, choose a unique name, and set the window styles for the class.

The following code fragment shows how to register the window class name "MyPrivateClass":

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1, MPARAM mp2);

HAB hab;

WinRegisterClass(hab,          /* Anchor block handle          */
    "MyPrivateClass",          /* Name of class being registered */
    ClientWndProc,             /* Window procedure for class    */
    CS_SIZEREDRAW |            /* Class style                   */
    CS_HITTEST,                /* Class style                   */
    0);                        /* Extra bytes to reserve        */
```

Summary

Following are the operating system functions and the structure used with window classes.

Table 3-3. Window Class Functions

Function Name	Description
WinQueryClassInfo	Returns window class information.
WinQueryClassName	Copies, into a buffer, the window class name as a null-terminated string.
WinRegisterClass	Registers a window class.
WinSubclassWindow	Subclasses the indicated window by replacing its window procedure with another window procedure.

Table 3-4. Window Class Structure

Structure Name	Description
CLASSINFO	Class-information structure.

Chapter 4. Window Procedures

Windows have an associated *window procedure*—a function that processes all messages sent or posted to a window. Every aspect of a window's appearance and behavior depends on the window procedure's response to the messages. This chapter explains how window procedures function, in general, and describes the default window procedure.

About Window Procedures

Every window belongs to a window class that determines which window procedure a particular window uses to process its messages. All windows of the same class use the same window procedure. For example, the operating system defines a window procedure for the frame window class (WC_FRAME), and all frame windows use that window procedure.

An application typically defines at least one new window class and an associated window procedure. Then, the application can create many windows of that class, all of which use the same window procedure. This means that the same piece of code can be called from several sources simultaneously; therefore, you must be careful when modifying shared resources from a window procedure.

Dialog procedures have the same structure and function as window procedures. The primary difference between a dialog procedure and a window procedure is the absence of a client window in the dialog procedure; that is, the controls in a dialog procedure are the immediate child windows of the frame, whereas the controls in a normal window are the *grandchildren* of the frame. This makes significant differences in the code between the two; for example, WinSendDlgItemMsg does not work from a client window if you pass the client window handle as the first parameter.

Structure of a Window Procedure

A window procedure is a function that takes 4 arguments and returns a 32-bit pointer. The arguments of a window procedure consist of a window handle, a ULONG message identifier, and two arguments, called *message parameters*, that are declared with the MPARAM data type. The system defines an MPARAM as a 32-bit pointer to a VOID data type (a generic pointer). The message parameters actually might contain any of the standard data types. The message parameters are interpreted differently, depending on the value of the message identifier. OS/2 2.0 includes several macros that enable the application to cast the information from the MPARAM values into the actual data type. SHORT1FROMMP, for example, extracts a 16-bit value from a 32-bit MPARAM.

The window-procedure arguments are described in the following table:

Table 4-1. Window Procedure Arguments

Argument	Description
hwnd	Handle of the window receiving the message.
msg	Message identifier. The message will correspond to one of the predefined constants (for example, WM_CREATE) defined in the system include files or be an application-defined message identifier. The value of an application-defined message identifier must be greater than the value of WM_USER, and less than or equal to *0xffff*.
mp1,mp2	Message parameters. Their interpretation depends on the particular message.

The return value of a window procedure is defined as an MRESULT data type. The interpretation of the return value depends on the particular message. Consult the description of each message to determine the appropriate return value.

Default Window Procedure

All windows in the system share certain fundamental behavior, defined in the default window-procedure function, WinDefWindowProc. The default window procedure provides the minimal functionality for a window. An application-defined window procedure should pass any messages it does not process to WinDefWindowProc for default processing.

Window-Procedure Subclassing

Subclassing enables an application to intercept and process messages sent or posted to a window before that window has a chance to process them. Subclassing most often is used to add functionality to a particular window or to alter a window's default behavior.

An application subclasses a window by using the WinSubclassWindow function to replace the window's original window procedure with an application-defined window procedure. Thereafter, the new window procedure processes any messages that are sent or posted to the window. If the new window procedure does not process a particular message, it must pass the message to the original window procedure, *not* to WinDefWindowProc, for default processing

Using Window Procedures

This section explains how to:

- Design a window procedure
- Associate a window procedure with a window class
- Subclass a window.

Designing a Window Procedure

The following code fragment shows the structure of a typical window procedure and how to use the message argument in a switch statement, with individual messages handled by separate case statements. Notice that each case returns a specific value for each message. For messages that it does not handle itself, the window procedure calls WinDefWindowProc.

```
MRESULT ClientWndProc(
HWND hwnd,
ULONG msg,
MPARAM mp1,
MPARAM mp2)
{
    /* Define local variables here, if required. */
    switch (msg) {
        case WM_CREATE:

    /* Initialize private window data.          */
        return (MRESULT) FALSE;

        case WM_PAINT:

    /* Paint the window.                        */
        return 0;

        case WM_DESTROY:

    /* Clean up private window data.            */
        return 0;

        default:
        break;
     }
     return WinDefWindowProc (hwnd, msg, mp1, mp2);
}
```

A dialog window procedure does not receive the WM_CREATE message; however, it does receive a WM_INITDLG message when all of its control windows have been created.

At the very least, a window procedure should handle the WM_PAINT message to draw itself. Typically, it should handle mouse and keyboard messages as well. Consult the descriptions of individual messages to determine whether your window procedure should handle them.

An application can call WinDefWindowProc as part of the processing of a message. In such a case, the application can modify the message parameters before passing the message to WinDefWindowProc or can continue with the default processing after performing its own operations.

Associating a Window Procedure with a Window Class

To associate a window procedure with a window class, an application must pass a pointer to that window procedure to the WinRegisterClass function. Once an application has registered the window procedure, the procedure automatically is associated with each new window created with that class.

The following code fragment shows how to associate the window procedure in the previous example with a window class:

```
HAB hab;
CHAR szClientClass[] = "My Window Class";

WinRegisterClass(hab,        /* Anchor-block handle  */
    szClientClass,           /* Class name           */
    ClientWndProc,           /* Pointer to procedure */
    CS_SIZEREDRAW,           /* Class style          */
    0);                      /* Window data          */
```

Subclassing a Window

To subclass a window, an application calls the WinSubclassWindow function, specifying the handle of the window to subclass and a pointer to the new window procedure. The WinSubclassWindow function returns a pointer to the original window procedure; the application can use this pointer to pass unprocessed messages to the original procedure.

The following code fragment subclasses a push button control window. The new window procedure generates a beep whenever the user clicks the push button.

```
    PFNWP pfnPushBtn;
    CHAR szCancel[] = "Cancel";
    HWND hwndClient;
    HWND hwndPushBtn;
        .
        .
        .
    /* Create a push button control.                 */
    hwndPushBtn = WinCreateWindow(
        hwndClient,      /* Parent-window handle       */
        WC_BUTTON,       /* Window class               */
        szCancel,        /* Window text                */
        WS_VISIBLE   |   /* Window style               */
        WS_SYNCPAINT |   /* Window style               */
        BS_PUSHBUTTON,   /* Button style               */
        50, 50,          /* Physical position          */
        70, 30,          /* Width and height           */
        hwndClient,      /* Owner-window handle        */
        HWND_TOP,        /* Z-order position           */
        1,               /* Window identifier          */
        NULL,            /* No control data            */
        NULL);           /* No presentation parameters */

    /* Subclass the push button control.             */
    pfnPushBtn = WinSubclassWindow(hwndPushBtn,
        SubclassPushBtnProc);
        .
        .
        .
}
    /* This procedure subclasses the push button.    */
    MRESULT EXPENTRY SubclassPushBtnProc(HWND hwnd,ULONG msg,MPARAM mp1, MPARAM mp2)
    {
        switch (msg) {

    /* Beep when the user clicks the push button.    */
            case WM_BUTTON1DOWN:
                DosBeep(1000, 250);
                break;

            default:
                break;
        }

    /* Pass all messages to the original window procedure. */
        return (MRESULT) pfnPushBtn(hwnd, msg, mp1, mp2);
    }
```

Summary

Following are the window-procedure functions and messages processed by the default window procedure.

Table 4-2. Window Procedure Functions

Function Name	Description
WinDefDlgProc	The default dialog procedure.
WinDefWindowProc	The default window procedure.
WinRegisterClass	Registers a window class.
WinSubclassWindow	Subclasses the indicated window by replacing its window procedure.

Table 4-3 (Page 1 of 2). Default Window Procedure Messages

Message	Description
WM_BUTTON1DBLCLK	Occurs when the user presses button 1 of the pointing device twice.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_BUTTON1UP	Occurs when the user releases pointer button 1.
WM_BUTTON2DBLCLK	Occurs when the user presses button 2 of the pointing device twice.
WM_BUTTON2DOWN	Occurs when the user presses pointer button 2.
WM_BUTTON2UP	Occurs when the user releases pointer button 2.
WM_BUTTON3DBLCLK	Occurs when the user presses button 3 of the pointing device twice.
WM_BUTTON3DOWN	Occurs when the user presses pointer button 3.
WM_BUTTON3UP	Occurs when the user releases pointer button 3.
WM_CALCVALIDRECTS	Sent to determine which areas of a window can be preserved if a window is sized and which can be redisplayed.
WM_CHAR	Occurs when the user presses a key.
WM_CLOSE	Sent to a frame window to indicate that the window is being closed by the user.
WM_CONTROLPOINTER	Sent to a control's owner window when the pointer moves over the control window, allowing the user to set the pointer.
WM_DDE_INITIATE	Sent by an application to one or more other applications to request initiation of a conversation.
WM_DDE_INITIATEACK	Sent by a server application in response to a WM_DDE_INITIATE message.
WM_FOCUSCHANGE	Occurs when the focus window is changed.
WM_HELP	Occurs when a control has a significant event to notify to its owner, or when a key stroke has been translated into a WM_HELP by an accelerator table.
WM_HITTEST	Sent to determine which window is associated with an input from the pointing device.
WM_MENUSELECT	Occurs when a menu item is selected.

Table 4-3 (Page 2 of 2). Default Window Procedure Messages

Message	Description
WM_MOUSEMOVE	Occurs when the pointing device pointer moves.
WM_PAINT	Occurs when a window needs repainting.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin DBCS conversion.
WM_QUERYFOCUSCHAIN	Requests the handle of a window in the focus chain.
WM_QUERYFRAMECTLCOUNT	Sent to the frame window in response to receipt of a WM_SIZE or WM_UPDATEFRAME message.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the window parameters.
WM_TIMER	Posted when a timer times out.
WM_TRANSLATEACCEL	Sent to the focus window when a WM_CHAR message occurs.

Chapter 5. Mouse and Keyboard Input

An OS/2 Presentation Manager application can accept input from both a mouse (or other pointing device) and the keyboard. This chapter explains how these *input events* should be received and processed.

About Mouse and Keyboard Input

Only one window at a time can receive keyboard input, and only one window at a time can receive mouse input; but they do not have to be the same window. All keyboard input goes to the window with the input focus, and, normally, all mouse input goes to the window under the mouse pointer.

System Message Queue

The operating system routes all keystrokes and mouse input to the system message queue, converting these input events into messages, and posts them, one at a time, to the proper application-defined message queues. An application retrieves messages from its queue and dispatches them to the appropriate window procedures, which process the messages.

Mouse and keyboard input events in the system message queue are strictly ordered so that a new event cannot be processed until all previous events are fully processed: the system cannot determine the destination window of an input event until then. For example, if a user types a command in one window, clicks the mouse to activate another window, then types a command in the second window, the destination of the second command depends on how the application handles the mouse click. The second command would go to the second window only if that window became active as a result of the mouse click.

It is important for an application to process all messages quickly to avoid slowing user interaction with the system. A message must be responded to immediately in the current thread, but the processing it initiates should be done asynchronously in another thread that has no windows in the desktop tree.

OS/2 can display multiple windows belonging to several applications at the same time. To manage input among these windows, the system uses the concepts of *window activation* and *keyboard focus.*

Window Activation

Although the operating system can display windows from many different applications simultaneously during a PM session, the user can interact with only one application at a time—the *active* application. The other applications continue to run, but they cannot receive user input until they become active.

To enable the user to easily identify the active application, the system activates all frames in the tree between HWND_DESKTOP and the window with input focus. That is, the system positions the active frame window above all other top-level windows on the screen. If the active window is a standard frame window, the window's title bar and sizing border are highlighted.

The user can control which application is active by clicking on a window or by pressing the Alt+Tab or Alt+Esc key combinations. An application can set the

active frame window by calling WinSetActiveWindow; it also can obtain the handle of the active frame window by using WinQueryActiveWindow.

When one window is deactivated and another activated, the system sends a WM_ACTIVATE message, first to the window being deactivated, then to the window being activated. The *fActive* parameter of the WM_ACTIVATE message is set to FALSE for the window being deactivated and set to TRUE for the window being activated. An application can use this message to track the activation state of a client window.

Keyboard Focus

The *keyboard focus* is a temporary attribute of a window; the window that has the keyboard focus receives all keyboard input until the focus changes to a different window. The system converts keyboard input events into WM_CHAR messages and posts them to the message queue of the window that has the keyboard focus.

An application can set the keyboard focus to a particular window by calling WinSetFocus. If the application does not use WinSetFocus to explicitly set the keyboard-focus window, the system sets the focus to the active frame window.

The following events occur when an application uses WinSetFocus to shift the keyboard focus from one window (the *original* window) to another (the *new* window):

1. The system sends the original window a WM_SETFOCUS message (with the *fFocus* parameter set to FALSE), indicating that that window has lost the keyboard focus.
2. The system then sends the original window a WM_SETSELECTION message, indicating that the window should remove the highlight from the current selection.
3. If the original (frame) window is being deactivated, the system sends it a WM_ACTIVATE message (with the *fActive* parameter set to FALSE), indicating that the window is no longer active.
4. The system then sends the new application a WM_ACTIVATE message (with *fActive* set to TRUE), indicating that the new application is now active.
5. If the new (main) window is being activated, the system sends it a WM_ACTIVATE message (with *fActive set to TRUE*), indicating that the main window is now active.
6. The system sends the new window a WM_SETSELECTION message, indicating that the window should highlight the current selection.
7. Finally, the system sends the new window a WM_SETFOCUS message (with *fFocus* set to TRUE), indicating that the new window has the keyboard focus.

If, while processing a WM_SETFOCUS message, an application calls WinQueryActiveWindow, that function returns the handle of the previously-active window until the application establishes a new active window. Similarly, if the application, while processing WM_SETFOCUS, calls WinQueryFocus, that function returns the handle of the previous keyboard-focus window until the application establishes a new keyboard-focus window. In other words, even though the system has sent WM_ACTIVATE and WM_SETFOCUS messages (with the *fActive* and *fFocus* parameters set to FALSE) to the previous windows, those windows are considered the active and focus windows until the system establishes new active and focus windows.

If the application calls WinSetFocus while processing a WM_ACTIVATE message, the system does not send a WM_SETFOCUS message (with *fFocus* set to FALSE), because no window has the focus.

A client window receives a WM_ACTIVATE message when its parent frame window is being activated or deactivated. The activation or deactivation message usually is followed by a WM_SETFOCUS message that specifies whether the client window is gaining or losing the keyboard focus. Therefore, if the client window needs to change the keyboard focus, it should do so during the WM_SETFOCUS message, not during the WM_ACTIVATE message.

Keyboard Messages

The system sends keyboard input events as WM_CHAR messages to the message queue of the keyboard-focus window. If no window has the keyboard focus, the system posts WM_CHAR messages to the message queue of the active frame window. Following are two typical situations in which an application receives WM_CHAR messages:

An application has a client window or custom control window, either of which can have the keyboard focus. If the window procedure for the client or control window does not process WM_CHAR messages, it should pass them to WinDefWindowProc, which will pass them to the owner. Dialog control windows, in particular, should pass unprocessed WM_CHAR messages to the WinDefDlgProc function, because this is how the user interface implements control processing for the Tab and Arrow keys.

An application window owns a control window whose window procedure can handle some, but not all, WM_CHAR messages. This is common in dialog windows. If the window procedure of a control in a dialog window cannot process a WM_CHAR message, the procedure can pass the message to the WinDefDlgProc function. This function sends the message to the control window's owner, which usually is a dialog frame window. The application's dialog procedure then receives the WM_CHAR message. This also is the case when an application client window owns a control window.

A WM_CHAR message can represent a key-down or key-up transition. It might contain a character code, virtual-key code, or scan code. This message also contains information about the state of the Shift, Ctrl, and Alt keys.

Each time a user presses a key, at least two WM_CHAR messages are generated: one when the key is pressed, and one when the key is released. If the user holds down the key long enough to trigger the keyboard repeat, multiple WM_CHAR key-down messages are generated. If the keyboard repeats faster than the application can retrieve the input events from its message queue, the system combines repeating character events into one WM_CHAR message and increments a count byte that indicates the number of keystrokes represented by the message. Generally, this byte is set to 1, but an application should check each WM_CHAR message to avoid missing any keystrokes.

An application can ignore the repeat count. For example, an application might ignore the repeat count on Arrow keys to prevent the cursor from skipping characters when the system is slow.

Message Flags

Applications decode WM_CHAR messages by examining individual bits in the flag word contained in the first message parameter (*mp1*) that the system passes with every WM_CHAR message. The type of flag word indicates the nature of the message. The system can set the bits in the flag word in various combinations. For example, a WM_CHAR message can have the KC_CHAR, KC_SCANCODE, and KC_SHIFT attribute bits all set at the same time. An application can use the following list of flag values to test the flag word and determine the nature of a WM_CHAR message:

Table 5-1 (Page 1 of 2). Keyboard Character Flags

Flag Name	Description
KC_ALT	Indicates that the Alt key was down when the message was generated.
KC_CHAR	Indicates that the message contains a valid character code for a key, typically an ASCII character code.
KC_COMPOSITE	In combination with the KC_CHAR flag, this flag indicates that the character code is a combination of the key that was pressed and the previous dead key. This flag is used to create characters with diacritical marks.
KC_CTRL	Indicates that the Ctrl key was down when the message was generated.
KC_DEADKEY	In combination with the KC_CHAR flag, this flag indicates that the character code represents a dead-key glyph (such as an accent). An application displays the dead-key glyph and does not advance the cursor. Typically, the next WM_CHAR message is a KC_COMPOSITE message, containing the glyph associated with the dead key.
KC_INVALIDCHAR	Indicates that the character is not valid for the current translation tables.
KC_INVALIDCOMP	Indicates that the character code is not valid in combination with the previous dead key.
KC_KEYUP	Indicates that the message was generated when the user released the key. If this flag is clear, the message was generated when the user pressed the key. An application can use this flag to determine key-down and key-up events.
KC_LONEKEY	In combination with the KC_KEYUP flag, this flag indicates that the user pressed no other key while this key was down.
KC_PREVDOWN	In combination with the KC_VIRTUALKEY flag, this flag indicates that the virtual key was pressed previously. If this flag is clear, the virtual key was not previously pressed.
KC_SCANCODE	Indicates that the message contains a valid scan code generated by the keyboard when the user pressed the key. The system uses the scan code to identify the character code in the current code page; therefore, most applications do not need the scan code unless they cannot identify the key that the user pressed. WM_CHAR messages generated by user keyboard input generally have a valid scan code, but WM_CHAR messages posted to the queue by other applications might not contain a scan code.
KC_SHIFT	Indicates that the Shift key was down when the message was generated.

Table 5-1 (Page 2 of 2). Keyboard Character Flags	
Flag Name	**Description**
KC_TOGGLE	Toggles on and off every time the user presses a specified key. This is important for keys like NumLock, which have an on or off state.
KC_VIRTUALKEY	Indicates that the message contains a valid virtual-key code for a key. Virtual keys typically correspond to function keys.

The *mp1* and *mp2* parameters of the WM_CHAR message contain information describing the nature of a keyboard input event, as follows:

- SHORT1FROMMP (*mp1*) contains the flag word.
- CHAR3FROMMP (*mp1*) contains the key-repeat count.
- CHAR4FROMMP (*mp1*) contains the scan code.
- SHORT1FROMMP (*mp2*) contains the character code.
- SHORT2FROMMP (*mp2*) contains the virtual key code.

An application window procedure should return TRUE if it processes a particular WM_CHAR message or FALSE if it does not. Typically, applications respond to key-down events and ignore key-up events.

The following sections describe the different types of WM_CHAR messages. Generally, an application decodes these messages by creating layers of conditional statements that discriminate among the different combinations of flag and code attributes that can occur in a keyboard message.

Key-Down or Key-Up Events

Typically, the first attribute that an application checks in a WM_CHAR message is the key-down or key-up event. If the KC_KEYUP bit of the flags word is set, the message is from a key-up event. If the flag is clear, the message is from a key-down event.

Repeat-Count Events

An application can check the key-repeat count of a WM_CHAR message to determine whether the message represents more than 1 keystroke. The count is greater than 1 if the keyboard is sending characters to the system queue faster than the application can retrieve them. If the system queue fills up, the system combines consecutive keyboard input events for each key into a single WM_CHAR message, with the key-repeat count set to the number of combined events.

Character Codes

The most typical use of WM_CHAR messages is to extract a character code from the message and display the character on the screen. When the KC_CHAR flag is set in the WM_CHAR message, the low word of *mp2* contains a character code based on the current code page. Generally, this value is a character code (typically, an ASCII code) for the key that was pressed.

Virtual-Key Codes

WM_CHAR messages often contain virtual-key codes that correspond to various function keys and direction keys on a typical keyboard. These keys do not correspond to any particular glyph code but are used to initiate operations. When the KC_VIRTUALKEY flag is set in the flag word of a WM_CHAR message, the high word of *mp2* contains a virtual-key code for the key.

Note: Some keys, such as the Enter key, have both a valid character code and a virtual-key code. WM_CHAR messages for these keys will contain character codes for both newline characters (ASCII 11) and virtual-key codes (VK_ENTER).

Scan Codes

A third possible value in a WM_CHAR message is the scan code of the key that was pressed. The scan code represents the value that the keyboard hardware generates when the user presses a key. An application can use the scan code to identify the physical key pressed, as opposed to the character code represented by the same key.

Accelerator-Table Entries

The system checks all incoming keyboard messages to see whether they match any existing accelerator-table entries (in either the system message queue or the application message queue). The system first checks the accelerator table associated with the active frame window; if it does not find a match, the system uses the accelerator table associated with the message queues. If the keyboard input event corresponds to an accelerator-table entry, the system changes the WM_CHAR message to a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message, depending on the attributes of the accelerator table. If the keyboard input event does not correspond to an accelerator-table entry, the system passes the WM_CHAR message to the keyboard-focus window.

Applications should use accelerator tables to implement keyboard shortcuts rather than translate command keystrokes. For example, if an application uses the F2 key to save a document, the application should create a keyboard accelerator entry for the F2 virtual key so that, when pressed, the F2 key generates a WM_COMMAND message rather than a WM_CHAR message.

Mouse Messages

Mouse messages occur when a user presses or releases one of the mouse buttons (a click) and when the mouse moves. All mouse messages contain the *x* and *y* coordinates of the mouse-pointer *hot spot* (relative to the coordinates of the window receiving the message) at the time the event occurs. The mouse-pointer hot spot is the location in the mouse-pointer bit map that the system tracks and recognizes as the position of the mouse pointer.

If a window has the CS_HITTEST style, the system sends the window a WM_HITTEST message when the window is about to receive a mouse message. Most applications pass WM_HITTEST messages on to WinDefWindowProc by default, so disabled windows do not receive mouse messages. Windows that specifically respond to WM_HITTEST messages can change this default behavior. If the window is enabled and should receive the mouse message, the WinDefWindowProc function (using the default processing for WM_HITTEST) returns the value HT_NORMAL. If the window is disabled, WinDefWindowProc returns HT_ERROR, in which case the window does not receive the mouse message.

The default window procedure processes the WM_HITTEST message and the *usHit* parameter in the WM_MOUSEMOVE message. Therefore, unless an application needs to return special values for the WM_HITTEST message or the *usHit* parameter, it can ignore them. One possible reason for processing the WM_HITTEST message is for the application to react differently to a mouse click in a disabled window.

The contents of the mouse-message parameters (*mp1* and *mp2*) are as follows:

- SHORT1FROMMP (*mp1*) contains the *x* position.
- SHORT2FROMMP (*mp1*) contains the *y* position.
- SHORT1FROMMP (*mp2*) contains the hit-test parameter.

Capturing Mouse Input

The operating system generally posts mouse messages to the window that is under the mouse pointer at the time the system reads the mouse input events from the system message queue. An application can change this by using the WinSetCapture function to route all mouse messages to a specific window or to the message queue associated with the current thread. If mouse messages are routed to a specific window, that window receives all mouse input until either the window releases the mouse or the application specifies another capture window. If mouse messages are routed to the current message queue, the system posts each mouse message to the queue with the *hwnd* member of the QMSG structure for each message set to NULL. Because no window handle is specified, the WinDispatchMsg function in the application's main message loop cannot pass these messages to a window procedure for processing. Therefore, the application must process these messages in the main loop.

Capturing mouse input is useful if a window needs to receive all mouse input, even when the pointer moves outside the window. For example, applications commonly track the mouse-pointer position after a mouse "button down" event, following the pointer until a "button up" event is received from the system. If an application does not call WinSetCapture for a window and the user releases the mouse button, the application does not receive the button-up message. If the application sets a window to capture the mouse and tracks the mouse pointer, the application receives the button-up message even if the user moves the mouse pointer outside the window.

Some applications are designed to require a button-up message to match a button-down message. When processing a button-down message, these applications call WinSetCapture to set the capture to their own window; then, when processing a matching button-up message, they call WinSetCapture, with a NULL window handle, to release the mouse.

Button Clicks

An application window's response to a mouse click depends on whether the window is active. The first click in an inactive window should activate the window. Subsequent clicks in the active window produce an application-specific action.

A common problem for an application that processes WM_BUTTON1DOWN or similar messages is failing to activate the window or set the keyboard focus. If the window processes WM_CHAR messages, the window procedure should call WinSetFocus to make sure the window receives the keyboard focus and is activated. If the window does not process WM_CHAR messages, the application should call WinSetActiveWindow to activate the window.

Mouse Movement

The system sends WM_MOUSEMOVE messages to the window that is under the mouse pointer, or to the window that currently has captured the mouse, whenever the mouse pointer moves. This is useful for tracking the mouse pointer and changing its shape, based on its location in a window. For example, the mouse pointer changes shape when it passes over the size border of a standard frame window.

All standard control windows use WM_MOUSEMOVE messages to set the mouse-pointer shape. If an application handles WM_MOUSEMOVE messages in some situations but not others, unused messages should be passed to the WinDefWindowProc function to change the mouse-pointer shape.

Using the Mouse and Keyboard

This section explains how to perform the following tasks:

- Determine the active status of a frame window.
- Check for a key-up or key-down event.
- Respond to a character message.
- Handle virtual-key codes.
- Handle a scan code.

Determining the Active Status of a Frame Window

The activated state of a window is a frame-window characteristic. The system does not provide an easy way to determine whether a client window is part of the active frame window. That is, the window handle returned by the WinQueryActiveWindow function identifies the active frame window rather than the client window owned by the frame window.

Following are two methods for determining the activated state of a frame window that owns a particular client window:

- Call WinQueryActiveWindow and compare the window handle it returns with the handle of the frame window that contains the client window, as shown in the following code fragment:

```
HWND hwndClient;
BOOL fActivated;

fActivated = (WinQueryWindow(hwndClient, QW_PARENT) ==
              WinQueryActiveWindow(HWND_DESKTOP));
```

- Each time the frame window is activated, the client window receives a WM_ACTIVATE message with the low word of the *mp2* equal to TRUE. When the frame window is deactivated, the client window receives a WM_ACTIVATE message with a FALSE activation indicator.

Checking for a Key-Up or Key-Down Event

The following code fragment shows how to decode a WM_CHAR message to determine whether it indicates a key-up event or a key-down event:

```
USHORT fsKeyFlags;

case WM_CHAR:  {
USHORT fsKeyFlags = SHORT1FROMMP(mp1);

if (fsKeyFlags & KC_KEYUP) {
        .
        . /* Perform key-up processing.   */
        .
} else {
        .
        . /* Perform key-down processing. */
        .
}

return;

}
```

Responding to a Character Message

The following code fragment shows how to respond to a character message:

```
USHORT fsKeyFlags;
UCHAR  uchChr1;

case WM_CHAR:
fsKeyFlags = (USHORT) SHORT1FROMMP(mp1);

if (fsKeyFlags & KC_CHAR) {

    /* Get the character code from mp2. */
    uchChr1 = (UCHAR) CHAR1FROMMP(mp2);
    .
    . /* Process the character.        */
    .

    return TRUE;
}
```

If the KC_CHAR flag is not set, the *mp2* parameter from CHAR1FROMMP still might contain useful information. If either the Alt key or the Ctrl key, or both, are down, the KC_CHAR bit is not set when the user presses another key. For example, if the user presses the **a** key when the Alt key is down, the low word of *mp2* contains the ASCII value for "a" (0x0061), the KC_ALT flag is set, and the KC_CHAR flag is clear. If the translation does not generate any valid characters, the **char** field is set to 0.

Handling Virtual-Key Codes

The following code fragment shows how to decode a WM_CHAR message containing a valid virtual-key code:

```
USHORT fsKeyFlags;

case WM_CHAR:
fsKeyFlags = (USHORT) SHORT1FROMMP(mp1);

if (fsKeyFlags & KC_VIRTUALKEY) {

    /* Get the virtual key from mp2.        */
    switch (SHORT2FROMMP(mp2)) {
     case VK_TAB:
         .
         . /* Process the TAB key.          */
         .
         return TRUE;
     case VK_LEFT:
         .
         . /* Process the LEFT key.         */
         .
         return TRUE;
     case VK_UP:
         .
         . /* Process the UP key.           */
         .
         return TRUE;
     case VK_RIGHT:
         .
         . /* Process the RIGHT key.        */
         .
         return TRUE;
     case VK_DOWN:
         .
         . /* Process the DOWN key.         */
         .
         return TRUE;
       .
       . /* Etc...                          */
       .
       default:
           return FALSE;
       }
    }
```

Handling a Scan Code

All WM_CHAR messages generated by keyboard input events have valid scan codes. WM_CHAR messages posted by other applications might or might not have valid scan codes. The following code fragment shows how to extract a scan code from a WM_CHAR message:

```
USHORT fsKeyFlags;
UCHAR  uchScanCode;

case WM_CHAR:
fsKeyFlags = (USHORT) SHORT1FROMMP(mp1);

if (fsKeyFlags & KC_SCANCODE) {

    /* Get the scan code from mp1.   */
    uchScanCode = CHAR4FROMMP(mp1);
    .
    . /* Process the scan code.      */
    .

    return (MRESULT) TRUE;
   }
```

Summary

Following are the OS/2 functions and messages used with activation and keyboard/mouse input.

Table 5-2. Mouse/Keyboard Functions

Function Name	Description
WinEnablePhysInput	Enables or disables queuing of physical input.
WinFocusChange	Changes the focus window.
WinGetKeyState	Returns the state of the key at the time the last message from the message queue was posted.
WinGetPhysKeyState	Returns the physical key state.
WinIsPhysInputEnabled	Returns the status of the hardware (*on/off*)
WinQueryActiveWindow	Returns the active window for HWND_DESKTOP or other parent window.
WinQueryCapture	Returns the handle of the window the pointer has captured.
WinQueryFocus	Returns the focus window; NULL if there is not focus window.
WinSetActiveWindow	Makes the frame window the active window.
WinSetCapture	Captures all pointing device messages.
WinSetFocus	Sets the focus window.
WinSetKeyboardStateTable	Gets or sets the keyboard state.

Table 5-3. Focus-Change and Activation Messages	
Message	**Description**
WM_ACTIVATE	Sent when a different window becomes the active window.
WM_FOCUSCHANGE	Occurs when the window having the focus is changed.
WM_QUERYFOCUSCHAIN	Requests the handle of a window in the focus chain.
WM_SETFOCUS	Occurs when a window is to lose or gain the input focus.
WM_SETSELECTION	Occurs when a window is selected or deselected.

Table 5-4. Mouse Messages	
Message	**Description**
WM_BUTTON1DBLCLK	Occurs when the user presses button 1 of the pointing device twice.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_BUTTON1UP	Occurs when the user releases pointer button 1.
WM_BUTTON2DBLCLK	Occurs when the user presses button 2 of the pointing device twice.
WM_BUTTON2DOWN	Occurs when the user presses pointer button 2.
WM_BUTTON2UP	Occurs when the user releases pointer button 2.
WM_BUTTON3DBLCLK	Occurs when the user presses button 3 on the pointing device twice.
WM_BUTTON3DOWN	Occurs when the user presses pointer button 3.
WM_BUTTON3UP	Occurs when the user releases pointer button 3.
WM_HITTEST	Sent to determine which window is associated with an input from the pointing device.
WM_MOUSEMOVE	Occurs when the pointing device pointer moves.

Table 5-5. Keyboard Messages	
Message	**Description**
WM_CHAR	Occurs when the user presses a key.
WM_COMMAND	Occurs when a control has a significant event to notify to its owner, or when a keystroke has been translated by an accelerator table into WM_COMMAND.

Chapter 6. Frame Windows

A *frame window* is the basic window used by most Presentation Manager applications to enable the user to perform manipulation functions. This chapter explains how to create and use frame windows in PM applications.

About Frame Windows

An application nearly always starts with a frame window to create a *composite window* (for example, a main window) that consists of the frame window, several frame-control windows, and a client window. The frame controls conform to the Common User Access (CUA) user interface guidlines. The frame window coordinates the actions of the frame controls and client window, enabling the composite window to act as a single unit.

Frame windows have the preregistered public window class WC_FRAME. The frame-window class, like the preregistered control classes, defines the appearance and behavior of the frame window.

Main Window

The *main window* of an application, typically, is composed of a frame window and a client window. The frame window usually includes control windows such as a title bar, system menu, menu bar (*action bar* or *menu* in user terminology), and scroll bars. Figure 6-1 is an example of a typical frame window.

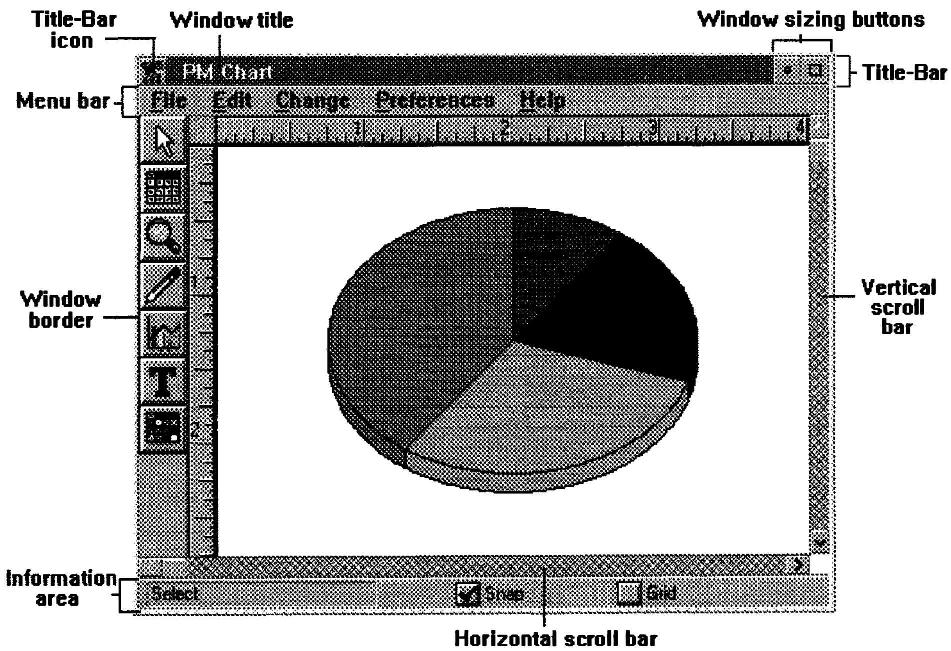


Figure 6-1. Typical Frame Window and Its Components

A frame window provides the standard services the user expects from a window—for example, moving, sizing, minimizing, and maximizing. The frame window receives input from the control windows (called *frame controls*) and sends messages to both the frame controls and the client window.

Frame Controls

When creating a frame window, an application also can create one or more frame controls as child windows of the frame window. Most frame windows contain at least a system menu and title bar. Other optional controls might include a menu bar and scroll bar as shown above.

An application can create a frame window with specified frame controls by calling WinCreateStdWindow with the appropriate frame-control flags.

The frame window owns the child frame-control windows, which can send notification messages that tell the frame window what the user is doing with the frame controls. For example, using a mouse, a user can move a window by clicking the title bar and dragging the window to a new position. The title-bar control responds to the click by sending a message to the frame window, notifying it of the user's request to move the window. Then the frame window tracks the mouse motion and moves the frame window and all of its child windows to the new position.

PM, rather than the application, handles the processing of the frame controls, thus providing the user a consistent interface for manipulating and interacting with windowed applications on the screen. Frame controls are described in individual chapters. For more information about control windows, see Chapter 7, "Control Windows" on page 7-1.

Client Window

Every main window has a *client window*, which is the window in which the application displays output and receives mouse and keyboard input from the user. What an application displays in the client window, how it displays it, and how it interprets input to the window are controlled by the client's application-defined window procedure.

An application creates the client window when it creates the frame window. The client window, which is specific to the application, is nearly always created using a *private window class* (a class registered by the application). Like a frame control, the client window is a child window and is owned by the frame window. This means, for example, that the client window is moved when the frame window moves, is clipped to the frame-window size, and is destroyed when the frame window is destroyed.

The relationship between the frame window and the client window allows the frame window to pass messages between other frame controls and the client window. For example, a client window can send a message to the frame window requesting that the frame window change the window title. The frame window, in turn, sends a message to the title-bar control, telling it to change the title of the window.

Additional Frame-Window Items

In addition to its frame controls, a frame window also can contain a sizing border and the minimize and maximize buttons (also known as maximize and minimize *icons*). These items are not frame controls, because the frame window draws and maintains them. (*Frame controls* are windows that draw and maintain themselves.)

The sizing border encloses the frame window and lets the user change the size of the window using a mouse. The minimize button, at the right end of the title bar, lets the user reduce the frame window to an icon. The maximize button, to the right of the minimize button, lets the user enlarge the window so that it fills the screen.

An application can add these items to a frame window by using the FCF_SIZEBORDER, FCF_MAXBUTTON, and FCF_MINBUTTON (or FCF_MINMAX) styles. (The FCF_MINMAX style adds both a maximize button and a minimize button.)

Frame-Control Identifiers

A frame window uses a set of standard constants to identify the frame controls and the client window. The *frame-control identifiers* all begin with the prefix FID_ and can be used in functions such as WinWindowFromID to uniquely identify a given control or the client window. The frame controls also use these identifiers in notification messages sent to the frame window. The following table describes the frame-control identifiers:

Table 6-1. Frame-Control Identifiers

Identifier	Description
FID_CLIENT	Identifies a client window.
FID_HORZSCROLL	Identifies a horizontal scroll bar.
FID_MENU	Identifies a menu.
FID_MINMAX	Identifies the minimize and maximize (*window-sizing*) buttons.
FID_SYSMENU	Identifies a system menu.
FID_TITLEBAR	Identifies a title bar.
FID_VERTSCROLL	Identifies a vertical scroll bar.

Frame-Window Creation

An application typically creates a frame window by using the WinCreateStdWindow function, which creates a frame window, a client window, and the specified frame controls. The application also can call WinCreateWindow with the WC_FRAME window class, which creates the frame window and controls but not the client window. To create the client, the application can call WinCreateWindow, specifying the original frame window as the parent and owner.

An application also can use a frame window to create a dialog window. For a dialog window, the frame window contains control windows but no client window. The application creates the dialog window by using the WinLoadDlg or WinCreateDlg functions. These functions require an appropriate dialog template from the application's resource-definition file. The dialog template specifies the styles and dimensions for the frame window and for the control windows that compose the dialog window.

Frame Window Controls and Styles

An application uses frame-control flags in the WinCreateStdWindow function to specify which frame controls to give to the frame window. Frame-control flags are constants that have the FCF_ prefix.

The frame-window class (WC_FRAME), like other public window classes, provides many class-specific window styles that applications can use to adapt the appearance and behavior of a frame window. To specify the frame-window styles, an application can use either frame-control flags or the frame-window style constants, which have the FS_ prefix. Each style constant has a corresponding frame-control flag. Both produce exactly the same styles in a frame window. Typically, if an application is creating a frame window that uses frame controls, the

application uses frame-control flags to specify the frame-window styles—if not, the application uses frame-style constants. An application can combine the frame-style constants with the standard window styles when creating a frame window.

When an application calls WinCreateStdWindow without setting any frame-control flags, the function creates a standard window that is invisible and behind all its sibling windows, that has a width and height of 0, and that is positioned at the lower-left corner of its parent window. After the call to WinCreateStdWindow returns, the application can use the WinSetWindowPos function to change the window's size, coordinates, z-order position, and visibility.

If an application calls WinCreateStdWindow with the FCF_SHELLPOSITION frame-control flag, the function creates the window so that it is in front of its sibling windows and has a standard size and coordinates determined by the system.

Frame-Window Resources

If an application specifies FCF_ACCELTABLE, FCF_ICON, FCF_MENU, FCF_STANDARD, FS_ACCELTABLE, FS_ICON, or FS_STANDARD when creating a frame window, the application must provide the resources to support the specified style. Failure to do so causes the window creation to fail. Depending on the style, a frame window might attempt to load one or more resources from the application's executable files.

The following table shows the frame-control flags and frame-window styles that require resources:

Table 6-2. Frame Window Flags and Styles Requiring Resources

Flag	Style	Description
FCF_ACCELTABLE	FS_ACCELTABLE	Requires an accelerator-table resource. The frame window uses the accelerator table to translate WM_CHAR messages to WM_COMMAND, WM_HELP, or WM_SYSCOMMAND messages.
FCF_ICON	FS_ICON	Requires an icon resource. The frame window draws the icon when the user minimizes the window.
FCF_MENU	FS_MENU	Requires a menu-template resource. A frame window uses the menu template to create a menu containing the commands and menus specified by the resource.
FCF_STANDARD	FS_STANDARD	Requires a menu-template resource (FCF_STANDARD only), an accelerator-table resource, and an icon resource.

You can use the resource compiler to add icon, menu, and accelerator-table resources to the application's executable file. Each resource must have a resource identifier that matches the resource identifier specified in the FRAMECDATA structure passed to the WinCreateWindow function, or in the *idResources* parameter of the WinCreateStdWindow function.

Note: For detailed information about icon, menu, and accelerator-table resources, see Chapter 26, "Mouse Pointers and Icons" on page 26-1, Chapter 11, "Menus" on page 11-1, and Chapter 22, "Keyboard Accelerators" on page 22-1 respectively.

The following sample code illustrates how to use WinCreateStdWindow to load and set up certain resources for a frame window. Normally the first step is to set up a header file defining the the IDs of the applicable resources:

```
#define ID_RESOURCE 001

#define IDM_OPTIONS  50
#define IDM_SHIFT    51
#define IDM_EXIT     52
```

Figure 6-2. Defining Resources for Header File

Then, make a resource (.RC) file, defining each resource:

```
/* Sample Resource */

#include <os2.h>

POINTER ID_RESOURCE  sampres.ico            /* Icon            */

ACCELTABLE ID_RESOURCE
BEGIN                                       /* Accelerator table */
     VK_F10,   IDM_SHIFT,   VIRTUALKEY
     VK_F3 ,   IDM_EXIT,    VIRTUALKEY
END

MENU ID_RESOURCE                            /* Menu            */
BEGIN
     SUBMENU "~Options", IDM_OPTIONS
      BEGIN
         MENUITEM "~Shift Colors\tF10", IDM_SHIFT
         MENUITEM "~Exit\tF3",          IDM_EXIT
      END
END
```

Figure 6-3. Defining Resources for Resource (.RC) File

When using WinCreateStdWindow with more than one resource, each resource can have the same ID, as in the above example (ID_RESOURCE or *1*), *but only if each resource is of a different type*. Resources of the same type must have unique IDs.

Use FCF flags to indicate what resources to load:

```
ULONG flFrameFlags=
                    FCF_TITLEBAR       |  /*  Title bar              */
                    FCF_SIZEBORDER     |  /*  Size border            */
                    FCF_MINMAX         |  /*  Min & Max buttons      */
                    FCF_SYSMENU        |  /*  System menu            */
                    FCF_SHELLPOSITION  |  /*  System size & position */
                    FCF_TASKLIST       |  /*  Add name to task list  */
                    FCF_ICON           |  /***Add icon               */
                    FCF_ACCELTABLE     |  /***Add accel. table       */
                    FCF_MENU           ;  /***Add menu               */
```

Figure 6-4. Using FCF Flags to Indicate What Resources to Load

Use 0 (or NULL) in the seventh parameter of WinCreateStdWindow to indicate that the resource is stored in the application file, as follows:

```
hwndFrame = WinCreateStdWindow(
    HWND_DESKTOP,       /* Parent is desktop window.           */
    WS_VISIBLE,         /* Make frame window visible.          */
    &flFrameFlags,      /* Frame controls                      */
    "ResSamClient",     /* Window class for client             */
    NULL,               /* No window title                     */
    WS_VISIBLE,         /* Make client window visible .        */
    (HMODULE) 0,        /* Resources in application module     */
    ID_RESOURCE,        /* Resource identifier                 */
    NULL);              /* Pointer to client window handle     */
```

Figure 6-5. Indicating that a Resource is Stored in the Application File

Following is the full listing of the sample program:

```
#define INCL_PM
#include <os2.h>

MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2);

int main(int argc, char *argv, char *envp)
{
    HWND hwndFrame;
    HWND hwndClient;
    HMQ  hmq;
    QMSG qmsg;
    HAB  hab;

    ULONG flFrameFlags=
                    FCF_TITLEBAR      |  /*  Title bar             */
                    FCF_SIZEBORDER    |  /*  Size Border           */
                    FCF_MINMAX        |  /*  Min & Max Buttons     */
                    FCF_SYSMENU       |  /*  System Menu           */
                    FCF_SHELLPOSITION |  /*  System size & position */
                    FCF_TASKLIST      |  /*  Add name to task list. */
                    FCF_ICON          |  /***Add icon.             */
                    FCF_ACCELTABLE    |  /***Add accelerator table. */
                    FCF_MENU;            /***Add menu.             */

    hab = WinInitialize(0);

    hmq = WinCreateMsgQueue(hab, 0);

    WinRegisterClass(
        hab,                    /* Anchor block handle           */
        "ResSamClient",         /* Name of class being registered */
        (PFNWP)ClientWndProc,   /* Window procedure for class    */
        CS_SIZEREDRAW |         /* Class style                   */
        CS_HITTEST,             /* Class style                   */
        0);                     /* Extra bytes to reserve        */

    hwndFrame = WinCreateStdWindow(
        HWND_DESKTOP,       /* Parent is desktop window.         */
        WS_VISIBLE,         /* Make frame window visible.        */
        &flFrameFlags,      /* Frame controls                    */
        "ResSamClient",     /* Window class for client           */
        NULL,               /* No window title                   */
        WS_VISIBLE,         /* Make client window visible .      */
        (HMODULE) 0,        /* Resources in application module   */
        ID_RESOURCE,        /* Resource identifier               */
        NULL);              /* Pointer to client window handle   */


    while (WinGetMsg(hab, &qmsg, 0, 0, 0))
         WinDispatchMsg(hab, &qmsg);

    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);

    return 0;
}
```

Figure 6-6 (Part 1 of 2). Sample Program for Loading Resources in a Frame Window

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2)
{
    RECTL rcl;
    HPS   hps;
    static LONG  lColor=CLR_RED;
    switch (msg) {

        case WM_PAINT:
              hps=WinBeginPaint(hwnd,(HPS) NULL, &rcl);     /* Get hps          */
              WinFillRect(hps,&rcl,lColor);                 /* Fill the window  */
              WinEndPaint(hps);                             /* Free hps         */
        return 0;

        case WM_COMMAND:

             switch (SHORT1FROMMP(mp1)) {
             case IDM_SHIFT:                                /* Shift selected   */
                if (lColor==CLR_RED) lColor=CLR_BLUE;       /* Change the       */
                else lColor=CLR_RED;                        /* color            */
                WinInvalidateRect(hwnd,(PRECTL)NULL,0UL);   /* Paint Window     */
                return 0;
             case IDM_EXIT:                                 /* Exit selected    */
                WinPostMsg(hwnd,WM_CLOSE,MPVOID,MPVOID);    /* Exit program.    */
                return 0;
             }

    }
    return WinDefWindowProc (hwnd, msg, mp1, mp2);
}
```

Figure 6-6 (Part 2 of 2). Sample Program for Loading Resources in a Frame Window

Frame-Window Class Data

An application can specify class-specific data for a frame window by passing to the WinCreateWindow function a pointer to the FRAMECDATA structure. The class-specific data contains the frame-control flags (FCF_ flags), resource-module handle, and resource identifier to be used when creating the frame window. The resource-module handle and the resource identifier specify where to find resources for the frame window.

Supplying class-specific data with WinCreateWindow is similar to using the WinCreateStdWindow function without creating a client window.

Frame-Window Data

Frame-window data specifies the state of the frame window at a given time. An application can retrieve the frame-window data by calling the WinQueryWindowUShort function. A frame window has the following state flags:

Table 6-3 (Page 1 of 2). Frame Window State Flags and Their Meanings

Flag	Description
FF_ACTIVE	Indicates that the frame window is active.
FF_DLGDISMISSED	Indicates that a dialog window has been dismissed by a call to the WinDismissDlg function.
FF_FLASHHILITE	Indicates that the frame window is flashing and its flash state is TRUE.
FF_FLASHWINDOW	Indicates that the frame window flashes as the result of either a call to the WinFlashWindow function or a WM_FLASHWINDOW message.

Table 6-3 (Page 2 of 2). Frame Window State Flags and Their Meanings

Flag	Description
FF_NOACTIVATESWP	Indicates that the system should do no z-ordering on this frame window.
FF_OWNERDISABLE	For a frame window that is part of a dialog window, this flag indicates whether the owner window was enabled or disabled when the dialog window was loaded.
FF_OWNERHIDDEN	Indicates that the frame window's owner window is hidden or minimized, in which case the frame window also is hidden.
FF_SELECTED	Indicates that the frame window has been selected.
FI_ACTIVATEOK	Indicates that the window can be activated.
FI_FRAME	Indicates that the window is a frame window.
FI_NOMOVEWITHOWNER	Indicates that the window should move when its owner window moves.
FI_OWNERHIDE	Indicates that the frame window should be hidden or shown as a result of its owner window being hidden, shown, minimized, or maximized.

Frame-Window Operation

The frame window maintains the size, position, and visibility of itself, its frame controls, and its client window. The frame window responds to user requests to move, size, minimize, maximize, and redraw itself. It also responds to requests to close (destroy) itself and to change the focus and activation state.

The frame window, when being moved or sized, maintains the position of each owned window relative to its owner window's lower-left corner.

Whenever the frame window redraws itself (for example, after being moved or sized), it draws the frame controls and then lets the application draw the client window. This order ensures that the rapidly drawn frame controls are drawn before the client window.

The order in which the frame controls are drawn depends on the z-order position of the controls. The following list specifies the z-order position of the frame controls (from top to bottom):

FID_SYSMENU
FID_TITLEBAR
FID_MENU
FID_VERTSCROLL
FID_HORZSCROLL
FID_CLIENT

Although an application can change the z-order position of any window, changing the relative positions of frame controls is not recommended.

When the user maximizes the frame window, the size of the frame window increases to the size of its parent window, plus an additional amount on each of its four sides equal to the width of its sizing border. A window always is clipped to its

parent window; a maximized standard frame window does not show its sizing border in its normal maximized position.

Frame controls owned by a frame window or windows owned by child windows of a frame window are destroyed automatically when the frame window processes the WM_DESTROY message.

Nonstandard Frame Windows

Although most applications use frame windows to create their main windows and dialog windows, they are not limited to frame windows. Applications can create nonstandard frame windows and still use the standard frame controls, such as the title bar and system menu, within the nonstandard windows.

An application can create a nonstandard frame window either by subclassing a frame window or by creating a private frame-window class. An application that subclasses a frame window can intercept the messages sent to the window and process them in new ways. An application that creates private frame-window classes essentially rewrites the frame-window procedure. In either case, by creating nonstandard frame windows, the application gains much more control over the arrangement of frame controls in the frame window.

The messages WM_FORMATFRAME, WM_UPDATEFRAME, and WM_CALCVALIDRECTS control the arrangement of frame controls for applications that subclass the frame-window procedure. By intercepting these messages, an application can rearrange the frame controls in a frame window.

To maintain the size and position of frame controls, an application that creates private frame-window classes can use the WinCreateFrameControls and WinCalcFrameRect functions. These functions provide capabilities that are similar to those provided by frame windows.

Default Frame-Window Behavior

The following table lists all the messages specifically handled by the window procedure of the predefined frame-window class (WC_FRAME) and describes how the window procedure responds to each message.

Table 6-4 (Page 1 of 3). Default Frame-Window Messages and Behavior

Message	Description
WM_ACTIVATE	Sets the highlighted state of the title bar or border so that it matches the frame window's activation state.
WM_BUTTON1DOWN	If the frame window is minimized, captures the mouse; otherwise, activates the frame window.
WM_BUTTON2DOWN	Activates the frame window.
WM_BUTTON3DOWN	Activates the frame window.
WM_BUTTON1UP	Processes messages from minimized window frames.
WM_BUTTON1DBLCLK	If the frame window is minimized, posts a WM_SYSCOMMAND message to itself; otherwise, activates the frame window.

Table 6-4 (Page 2 of 3). Default Frame-Window Messages and Behavior	
Message	**Description**
WM_CALCVALIDRECTS	If the frame window has no client window or if the client window has the CS_SIZEREDRAW style, returns the CVR_REDRAW flag to invalidate the entire window.
WM_CLOSE	If the frame window has a client window, passes this message to the client; otherwise, returns the result of the WinDefWindowProc function.
WM_CREATE	Creates the specified frame controls by calling the WinCreateFrameControls function. Also creates any accelerator tables, loads icons, and adds itself to the Window List. These actions depend on the frame-window styles and frame-control flags specified for the window.
WM_DESTROY	If the focus is held by a child window of the frame window, sets the focus to the frame window's parent window, destroys any owned windows or child windows, destroys any icons created by using the FS_ICON style, and destroys any accelerator tables created by using the FS_ACCELTABLE style.
WM_ENABLE	Returns the result of the WinDefWindowProc function.
WM_ERASEBACKGROUND	Returns TRUE, signaling that the window should erase the client-window area. The frame window sends this message to itself during WM_PAINT processing.
WM_FORMATFRAME	Calculates the sizes and positions of the frame controls and the client window.
WM_HITTEST	If the frame window is minimized and disabled, returns HT_ERROR; otherwise, returns TF_MOVE.
WM_MINMAXFRAME	If the frame window has a client window, passes this message to the client window; otherwise, passes this message to the WinDefWindowProc function.
WM_MOUSEMOVE	Determines the correct mouse pointer to use and returns the result of WinDefWindowProc.
WM_PAINT	If the frame window is minimized, sends WM_QUERYICON and WM_ERASEBACKGROUND to itself and draws the icon; otherwise, paints the control windows, sends a WM_ERASEBACKGROUND message to the client window, and paints the client window.
WM_QUERYTRACKINFO	Starts track-move processing of the title-bar control window.
WM_SHOW	Returns the result of WinDefWindowProc.
WM_SIZE	Sends a WM_FORMATFRAME message to itself.

Table 6-4 (Page 3 of 3). Default Frame-Window Messages and Behavior	
Message	**Description**
WM_SYSCOMMAND	If the frame window has captured the mouse, ignores the system command; otherwise, uses one of the following commands: SC_APPMENU, SC_CLOSE, SC_MOVE, SC_NEXT, SC_NEXTFRAME, SC_RESTORE, SC_SIZE, SC_SYSMENU, SC_TASKMANAGER.
WM_UPDATEFRAME	Reformats and updates the appearance of the frame window. Sent after a frame control has been added to or removed from the frame window.

Using Frame Windows

This section explains how to:

- Create a main window
- Retrieve a frame-control handle.

Creating a Main Window

An application can create a main window by using the WinCreateStdWindow function. The following code fragment creates a typical main window—a frame window that has a system menu, title bar, menu, vertical and horizontal scroll bars, minimize and maximize (window-sizing) buttons, and a sizing border:

```
#define IDM_MENU 1

HWND hwndFrame;

ULONG flFrameControlFlags =
FCF_SYSMENU    | FCF_TITLEBAR | FCF_SIZEBORDER |
FCF_MENU       | FCF_MINMAX   | FCF_HORZSCROLL |
FCF_VERTSCROLL | FCF_SHELLPOSITION;

hwndFrame = WinCreateStdWindow(
HWND_DESKTOP,           /* Frame-window parent      */
WS_VISIBLE,             /* Make window visible      */
&flFrameControlFlags,   /* Frame-control flags      */
"MyClass",              /* Client-window class      */
"Main Window",          /* Window title             */
0,                      /* No client-window styles  */
(HMODULE)NULL,          /* App. module has resources */
IDM_MENU,               /* Resource ID              */
0);                     /* Client-window handle     */
```

An application also can create a *standard* main window by creating a frame window with the FCF_STANDARD flag. The application must include icon, menu, and accelerator-table resources if it uses the FCF_STANDARD flag.

The application creates the standard window by using the WinCreateStdWindow function, as shown in the following code fragment:

```
#define IDM_RESOURCES 1

HWND hwndFrame;

/* Set the frame-control flags.                                  */
ULONG flFrameControlFlags = FCF_STANDARD;

/* Create the standard main window.                              */
hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE,
&flFrameControlFlags,
"MyClass", "Main Window", 0, (HMODULE) NULL,
IDM_RESOURCES, 0);
```

Another way to create a main window and its frame controls is to use the WinCreateWindow function to create the frame window and the frame controls, then call WinCreateWindow again to create the client window. One advantage of this approach is that, when creating the frame window, the application can specify the window's initial size and position. The following code fragment illustrates this approach:

```
#define ID_RESOURCES 1
#define ID_FRAME     1

ULONG flFrameControlFlags =
FCF_ACCELTABLE | FCF_ICON       | FCF_MENU    |
FCF_MINMAX     | FCF_SIZEBORDER | FCF_SYSMENU |
FCF_TASKLIST   | FCF_TITLEBAR;

FRAMECDATA fcdata;
HWND hwndFrame;
HWND hwndClient;
SWP swp;

fcdata.cb = sizeof(FRAMECDATA);
fcdata.flCreateFlags = flFrameControlFlags;
fcdata.hmodResources = (HMODULE) NULL;
fcdata.idResources    = ID_RESOURCES;

/* Create the frame and client windows.            */
hwndFrame = WinCreateWindow(
HWND_DESKTOP,          /* Frame-window parent       */
WC_FRAME,              /* Frame-window class        */
"Main Window",         /* Window title              */
0,                     /* Initially invisible       */
0,0,0,0,               /* Size and position = 0     */
NULL,                  /* No owner                  */
HWND_TOP,              /* Top z-order position      */
ID_FRAME,              /* Frame-window ID           */
&fcdata,               /* Pointer to class data     */
NULL);                 /* No presentation parameters */

hwndClient = WinCreateWindow(
hwndFrame,             /* Client-window parent      */
"MyClass",             /* Client-window class       */
NULL,                  /* No title for client window */
0,                     /* Initially invisible       */
0,0,0,0,               /* Size and position = 0     */
hwndFrame,             /* Owner is frame window     */
HWND_BOTTOM,           /* Bottom z-order position   */
FID_CLIENT,            /* Standard client-window ID */
NULL,                  /* No class data             */
NULL);                 /* No presentation parameters */

.
. /* Continue with initialization.                 */
.

/* Set the size and position of the frame window.   */
WinQueryWindowPos(HWND_DESKTOP, &swp);
WinSetWindowPos(hwndFrame, HWND_TOP, swp.x, swp.cy / 2,
swp.cx, swp.cy / 2, SWP_MOVE | SWP_SIZE);

/* Set the size and position of the client window.  */
WinQueryWindowPos(hwndFrame, &swp);
WinSetWindowPos(hwndClient, HWND_TOP, SV_CXSIZEBORDER,
SV_CYSIZEBORDER - 1, swp.cx - SV_CXSIZEBORDER * 2,
(swp.cy - SV_CYSIZEBORDER * 2) + 1, SWP_MOVE | SWP_SIZE);

/* Make the frame and client windows visible.       */
WinShowWindow(hwndFrame, TRUE);
WinShowWindow(hwndClient, TRUE);
```

Retrieving a Frame Handle

An application can retrieve a frame-control handle by using the WinWindowFromID function. The following code fragment retrieves the handle of a title-bar control:

```
HWND hwndTitleBar,hwndFrame;

hwndTitleBar = WinWindowFromID(hwndFrame, FID_TITLEBAR);
```

Given a frame-control handle, an application can retrieve its parent frame-window handle by using the WinQueryWindow function:

```
HWND hwndFrame,hwndTitleBar;

hwndFrame = WinQueryWindow(hwndTitleBar, QW_PARENT);
```

By using identifiers to identify frame controls, rather than using window classes, an application can create its own controls to replace the predefined controls.

Summary

Following are the OS/2 functions, structures, and messages used with frame windows.

Table 6-5. Frame-Window Functions

Function Name	Description
WinCalcFrameRect	Calculates a client rectangle from a frame rectangle or a frame rectangle from a client rectangle.

Table 6-6. Frame-Window Structures

Structure Name	Description
FRAMECDATA	Frame-control data structure.
HSVWP	Frame window repositioning handle.

Table 6-7 (Page 1 of 3). Frame-Window Messages

Message	Description
WM_ACTIVATE	Occurs when an application causes the activation or deactivation of a window.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_BUTTON2DOWN	Occurs when the user presses pointer button 2.
WM_BUTTON3DOWN	Occurs when the user presses pointer button 3.
WM_BUTTON1UP	Occurs when the user releases point button 1.

Table 6-7 (Page 2 of 3). Frame-Window Messages

Message	Description
WM_CALCVALIDRECTS	Sent to determine which areas of a window can be preserved and which can be displayed when a window is sized.
WM_CLOSE	Sent to a frame window to indicate that the user is closing the window.
WM_CREATE	Occurs when the application requests creation of a window.
WM_DESTROY	Occurs when an application requests destruction of a window.
WM_ENABLE	Sets the enable state of a window.
WM_ERASEBACKGROUND	Causes a client window to be filled with the background, if appropriate.
WM_FLASHWINDOW	Occurs when an application has issued a WinFlashWindow call.
WM_FOCUSCHANGE	Occurs when the window possessing the focus is changed.
WM_FORMATFRAME	Sent to a frame window to calculate the sizes and positions of all the frame controls and the client window.
WM_HITTEST	Sent to determine which window is associated with an input from the pointing device.
WM_MINMAXFRAME	Sent to a frame window that is being minimized, maximized, or restored.
WM_MOUSEMOVE	Occurs when the pointing device pointer moves.
WM_NEXTMENU	Occurs when either the beginning or the end of the menu is reached using the cursor control keys.
WM_PAINT	Occurs when a window needs painting.
WM_QUERYFRAMECTLCOUNT	Sent to the frame window in response to the receipt of a WM_SIZE or WM_UPDATEFRAME message.
WM_QUERYFRAMEINFO	Enables an application to query information about frame windows.
WM_QUERYICON	Sent to a frame window to query its associated icon.
WM_QUERYTRACKINFO	The frame control and title bar generate this message after receiving a WM_TRACKFRAME message.
WM_SETACCELTABLE	Establishes the window accelerator table to be used for translation when the window is active.
WM_SETBORDERSIZE	Sent to the frame window to change the width and height of the border.
WM_SETICON	Sent to a frame window to set its associated icon.

Table 6-7 (Page 3 of 3). Frame-Window Messages	
Message	**Description**
WM_SHOW	Occurs when a window's WS_VISIBLE state is changing.
WM_SIZECLIPBOARD	Sent when the clipboard contains a data handle for the CFI_OWNERDISPLAY format, and the clipboard application window has changed size.
WM_SYSCOMMAND	Occurs when a control has a significant event to notify to its owner or when a keystroke has been translated by an accelerator table into a WM_SYSCOMMAND message.
WM_TRACKFRAME	Sent to a window whenever it is to be moved or sized.
WM_TRANSLATEACCEL	Sent to the focus window whenever a WM_CHAR message occurs.
WM_UPDATEFRAME	Sent by an application after frame controls have been added or removed from the window frame.
WM_WINDOWPOSCHANGED	Sent to the window procedure of the window whose position is changed.

Chapter 7. Control Windows

A *control window* is a window that an application uses in conjunction with another window to carry out simple input and output tasks. This chapter describes how to create and use control windows in PM applications.

About Control Windows

Control windows are used most often as part of a frame or dialog window, but they also can be used in a client window. An application can create control windows in a frame window by using frame-control flags in the WinCreateStdWindow function, or it can create control windows individually by calling the WinCreateWindow function.

Including control windows in a dialog window requires the use of a *dialog template*, which is a data structure that describes a dialog window and its control windows. The system uses the data in the dialog template to create the dialog window and control windows. An application can create a dialog template at run time, or it can use the system resource compiler to create a dialog-template resource.

The operating system provides many types of predefined control windows. An application can create a control of a particular type by specifying the appropriate control-window class name, either in the WinCreateWindow function or in a dialog template. The following is a list of the predefined control-window classes:

Table 7-1 (Page 1 of 2). Control Window Classes

Class name	Description
WC_BUTTON	Consists of buttons and boxes the user can select by clicking the pointing device or using the keyboard.
WC_COMBOBOX	Creates a combination-box control, which combines a list-box control and an entry-field control. It allows the user to enter data by typing in the entry field or choosing from a list in the list box.
WC_CONTAINER	Creates a control for the user to group objects in a logical manner. A container can display those objects in various formats or views. The container control supports drag and drop so the user can place information in a container by simply dragging and dropping.
WC_ENTRYFIELD	Consists of a single line of text that the user can edit.
WC_FRAME	A composite window class that can contain child windows of many of the other window classes.
WC_LISTBOX	Presents a list of text items from which the user can make selections.
WC_MENU	Presents a list of items that can be displayed horizontally as action bars, or vertically as pull-down menus. Menus usually are used to provide a command interface to applications.
WC_NOTEBOOK	Creates a control for the user that is displayed as a number of pages. The top page is visible, and the others are hidden, with their presence being indicated by a visible edge on each of the back pages.
WC_SCROLLBAR	Consists of window scroll bars that let the user request to scroll the contents of an associated window.

Class name	Description
WC_SLIDER	Creates a control that is usable for producing approximate (analog) values or properties. Scroll bars were used for this function in the past, but the slider provides a more flexible method of achieving the same result, with less programming effort.
WC_SPINBUTTON	Creates a control that presents itself to the user as a scrollable ring of choices, giving the user quick access to the data. The user is presented only one item at a time, so the spin button should be used with data that is intuitively related.
WC_STATIC	Simple display items that do not respond to keyboard or pointing device events.
WC_TITLEBAR	Displays the window title or caption and lets the user move the window's owner.
WC_VALUESET	Creates a control similar in function to the radio buttons but provides additional flexibility to display graphical, textual, and numeric formats. The values set with this control are mutually exclusive.

Table 7-1 (Page 2 of 2). Control Window Classes

A control window is always owned by another window, usually a frame or dialog window. This relationship is important because a control window sends WM_CONTROL messages to its owner whenever an input event occurs in the control window. Each WM_CONTROL message includes the identifier of the control window in which the event occurred and a notification code that specifies the nature of the event. An application specifies a control window's ID either in the WinCreateWindow function or in a dialog template. Each ID must be unique.

Control windows are like other predefined window classes in that they respond to standard window-management messages and functions, such as WinSetWindowText and WinShowWindow.

All control-window classes have a set of specific messages they send and receive. The summary at the end of this chapter lists the messages that all control windows have in common.

The system paints most control windows synchronously—that is, it redraws a control window as soon as any part of that window becomes invalid.

Using Control Windows

An application can use control windows in a dialog window, standard frame window, or client window. The following sections describe how to use control windows in an application.

Using Control Windows in a Dialog Window

To use a control window in a dialog window, an application specifies the control in a dialog template in the application's resource-definition file. A dialog template typically includes several control windows. When the application loads the dialog-template resource and displays the dialog window, the system automatically displays the control windows as part of the dialog window.

An application can send messages, through the dialog-window procedure, to a control window to change its state. The control window sends notification messages to the dialog-window procedure. The content of a notification message depends on the type of control window.

Using Control Windows in a Non-Dialog Window

To use a control window in a non-dialog window, an application must call the WinCreateWindow function, using the appropriate window class name. An application usually specifies one of its client windows as the owner of the control window. Therefore, the client-window procedure receives notification messages from the control window. In cases where a control is owned by the frame window (such as a menu control), the notification messages to the frame window are passed to the client window.

Creating a Custom Control Window

The operating system provides the following three ways to create custom control windows:

- Use ownerdraw list boxes and menus or buttons.
- Subclass an existing control-window class.
- Register and implement a window class from scratch.

List boxes and menus can have an *ownerdraw* style, and buttons can have a user-button style, which cause the system to send a message to the owner of the ownerdraw control whenever the control must be drawn. (If the owner is a frame window, it sends these messages on to its client windows for handling by the client window procedure.) This feature lets an application alter the appearance of a control window. For menus and list boxes, the owner window draws the items within the control, and the system draws the outline of the control. For buttons, the user-button style affects the drawing of the entire control.

Subclassing an existing control window is an easy way to create a custom control. The subclass procedure can alter selected behavior of the control window by processing only those messages that affect the selected behaviors. All other messages pass to the original control-window procedure.

The techniques for defining a custom control-window class are the same as those used for creating a client-window class. When you create a custom control-window class, be sure the window procedure can send and receive the messages listed in Table 7-2 on page 7-5 and Table 7-3 on page 7-5.

If an application creates a private control-window class, the name of the private class could be used in the dialog template, just like a predefined window-class constant. For example, if an application defines and registers a window class called "MyControlClass", it could create a dialog window that contains that type of control window by using the following resource definition:

```
DLGTEMPLATE IDD_CUSTOM_TEST
BEGIN
DIALOG "", IDD_CUSTOM_TEST, 1, 1, 126, 130, FS_DLGBORDER, 0
BEGIN
    CONTROL "This is Text", IDD_TITLE,
          37, 107, 56, 12,
          WC_STATIC,
          SS_TEXT | DT_CENTER | DT_TOP | DT_WORDBREAK
          | WS_VISIBLE
    CONTROL "Custom Control", IDD_CUSTOM,
          33, 68, 64, 13,
          "MyControlClass",
          WS_VISIBLE
    CONTROL "Okay", DID_OK,
          57, 10, 24, 14,
          WC_BUTTON,
          BS_PUSHBUTTON | BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
    END
END
```

Summary

Following are the OS/2 messages used with control windows.

Table 7-2. Messages Received by a Control Window

Message	Description
WM_ADJUSTWINDOWPOS	Sent by WinSetWindowPos to enable the window to adjust its new position or size when it is about to be moved.
WM_QUERYDLGCODE	Sent by the dialog manager to identify the type of control, to determine what kinds of messages the control understands, and to determine whether an input message may be processed by the dialog manager or passed down to the control.

Table 7-3. Messages Generated by a Control Window to its Owner

Message	Description
WM_COMMAND	Occurs when a control has a significant event to notify to its owner, or when a keystroke has been translated by an accelerator table.
WM_CONTROLPOINTER	Sent to a control's owner window when the pointing device pointer moves over the control window, allowing the owner to set the pointer.
WM_HELP	Occurs when a control has a significant event to notify to its owner, or when a keystroke has been translated into a WM_HELP message by an accelerator table.
WM_SYSCOMMAND	Occurs when a control has a significant event to notify to its owner, or when a keystroke has been translated into a WM_SYSCOMMAND message by an accelerator table.

Chapter 8. Button Controls

A *button* is a type of control window used to initiate an operation or to set the attributes of an operation. This chapter describes how to create and use buttons in PM applications.

About Button Controls

A button control can appear alone or with a group of other buttons. When buttons are grouped, the user can move from button to button within the group by pressing the Arrow keys. The user also can move among groups by pressing the Tab key.

A user can select a button by clicking it with the mouse, pressing the spacebar when the button has the keyboard focus, or sending a BM_CLICK message. In most cases, a button changes its appearance when selected.

A button control is always owned by another window, usually a dialog window or an application's client window. A button control posts WM_COMMAND messages or sends WM_CONTROL notification messages to its owner when a user selects the button. The owner window receives messages from a button control and can send messages to the button to alter its position, appearance, and enabled/disabled state.

To use a button control in a dialog window, an application specifies the control in a dialog template in the application's resource-definition file. The application processes button messages in the dialog-window procedure.

An application creates a button control in a client window by calling WinCreateWindow, specifying a window class of WC_BUTTON, and identifying the client window as the owner of the button control.

Button Types

There are four main *types* of buttons: push buttons, radio buttons, check boxes, and three-state check boxes. A button's type determines how the button looks and behaves.

A radio button, check box, or three-state check box *controls* an operation; a push button *initiates* an operation. For example, a user might set printing options (such as paper size, print quality, and printer type) in a print-command dialog window containing an array of radio buttons and check boxes. After setting the options, the user would select a push button to tell an application that printing should begin (or be canceled). Then, the application would query the state of each check box and radio button to determine the printing parameters.

A *push button* is a rectangular window that contains a text string, as shown in Figure 8-1 on page 8-2. Typically, an application uses a push button to let the user start or stop an operation.

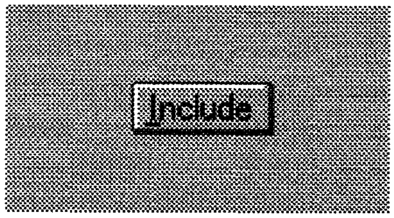


Figure 8-1. Push Button in a Dialog Box

When selected, a push button control posts a WM_COMMAND message to its owner window.

A *radio button* is a window with text displayed to the right of a small circular indicator. Each time the user selects a radio button, that button's state toggles between *selected* and *unselected.* This state remains until the next time the user selects the button. An application typically uses radio buttons in groups, as shown in Figure 8-2.

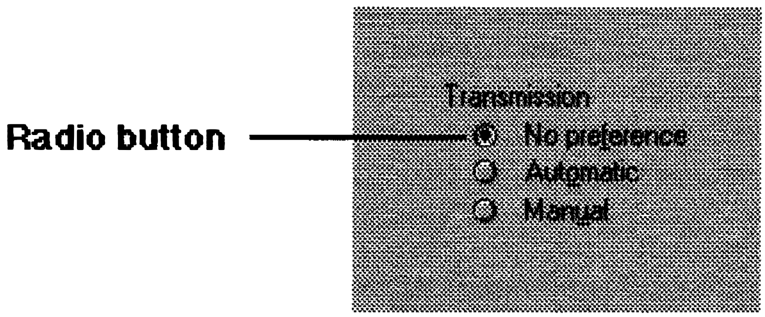


Figure 8-2. Radio Buttons in a Dialog Box

Within a group, usually one button is selected by default, and the user can move the selection to another button by using the cursor keys; however, only one button can be selected at a time. Radio buttons are appropriate if an *exclusive* choice is required from a fixed list of related options. For example, applications often use radio buttons to allow the user to select the screen foreground and background colors. A radio-button control sends WM_CONTROL messages to its owner window.

Check boxes are similar to radio buttons, except that they can offer *multiple-choice* selection as well as individual choice. Figure 8-3 offers the user a fixed list of choices, with the option of selecting more than one, or even all.

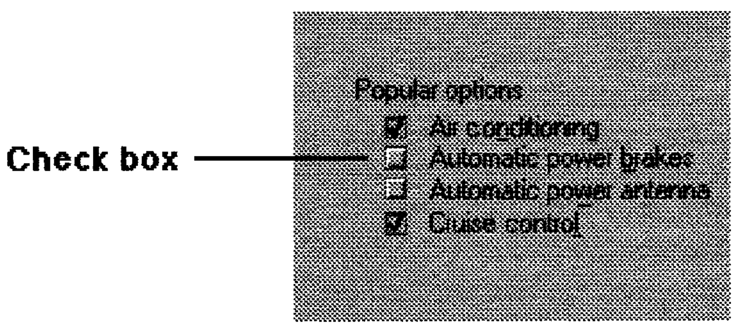


Figure 8-3. Check Boxes in a Dialog Box

Check boxes also toggle application features *on* or *off*. For example, a word processing application might use a check box to let the user turn word wrapping on or off. A check-box control sends WM_CONTROL messages to its owner window.

Three-state check boxes are similar to check boxes, except that they can be displayed in *halftone* as well as selected and unselected. An application might use the halftone state to indicate that, currently, the checkbox is not selectable. A three-state check-box control sends WM_CONTROL messages and posts WM_COMMAND messages to its owner window.

In addition to using the four predefined button-control types, an application can create button controls that appear as defined by the owner window. When they must be drawn or highlighted, these button controls send WM_CONTROL messages with BN_PAINT as the notification code to their owner windows.

Button Styles

The following table describes the button styles an application can use when creating button controls:

Table 8-1 (Page 1 of 3). Button Styles

Style	**Description.**
BS_3STATE	Creates a three-state check box (see also BS_CHECKBOX). When the user selects the check box, it sends a WM_CONTROL message to the owner window. The owner should set the check box to the appropriate state: selected, unselected, or halftone.
BS_AUTO3STATE	Creates an auto-three-state check box (see also BS_CHECKBOX). When the user selects the check box, the system automatically sets the check box to the appropriate state: selected, unselected, or halftone.
BS_AUTOCHECKBOX	Creates an auto-check box (see also BS_CHECKBOX). The system automatically toggles the check box between the selected and unselected states each time the user selects the box.
BS_AUTORADIOBUTTON	Creates an auto-radio button (see also BS_RADIOBUTTON). When the user selects an auto-radio button, the system automatically selects the button and removes the selection from the other auto-radio buttons in the group.
BS_CHECKBOX	Creates a check box—a small square that has text displayed to its right. When the user selects a check box, the check box sends a WM_CONTROL message to the owner window. The owner window should toggle the check box between selected and unselected states.

Style	Description.
BS_DEFAULT	Creates a push button that has a heavy black border. The user can select this push button by pressing the spacebar. This style is useful for letting the user quickly select the most likely set of options in a dialog window. This style is valid only in combination with the BS_PUSHBUTTON style or the PUSHBUTTON statement in a resource-definition file.
BS_HELP	Creates a push button that posts a WM_HELP message (instead of a WM_COMMAND message) to its owner window when the user selects the button. This style is valid only in combination with the BS_PUSHBUTTON style or the PUSHBUTTON statement in a resource-definition file.
BS_NOCURSORSELECT	Creates an auto-radio button that will not be selected automatically when the user moves the cursor to the button using the cursor-movement keys. This style is valid only in combination with the BS_AUTORADIOBUTTON style or the AUTORADIOBUTTON statement in a resource-definition file.
BS_NOBORDER	Creates a push button that has no border. This style is valid only in combination with the BS_PUSHBUTTON style or the PUSHBUTTON statement in a resource-definition file.
BS_NOPOINTERFOCUS	Creates a radio button or check box that does not receive the keyboard focus when the user selects it. This style is valid in combination with the BS_AUTORADIOBUTTON, BS_RADIOBUTTON, BS_3STATE, BS_AUTO3STATE, BS_AUTOCHECKBOX, and BS_CHECKBOX styles, or the AUTORADIOBUTTON, RADIOBUTTON, AUTOCHECKBOX, or CHECKBOX statements in a resource-definition file.
BS_PUSHBUTTON	Creates a push button—a round-cornered rectangle with text displayed inside it. When selected, the push button posts a WM_COMMAND message to its owner window.

Table 8-1 (Page 2 of 3). Button Styles

Table 8-1 (Page 3 of 3). Button Styles	
Style	**Description.**
BS_RADIOBUTTON	Creates a radio button—a small circle that has text displayed to its right. Radio buttons usually are used in groups of related, but exclusive, choices. When the user selects a radio button, the button sends a WM_CONTROL message to its owner window. The user should select the button and remove the selection from the other radio buttons in the group.
BS_SYSCOMMAND	Creates a button that posts a WM_SYSCOMMAND message (instead of a WM_COMMAND message) to the owner window when the user selects the button. This style is valid only in combination with the BS_PUSHBUTTON style or the PUSHBUTTON statement in a resource-definition file.
BS_USERBUTTON	Creates a user-defined button that sends a WM_CONTROL message to the owner window when the button needs to be drawn, highlighted, or disabled. A user-defined button also posts WM_COMMAND messages to the owner window when the user selects the button.

Default Button Behavior

Following are the messages processed by the predefined button-control window class (WC_BUTTON). Each message is described in terms of how a button control responds to that message.

Table 8-2 (Page 1 of 2). Messages Processed by the WC_BUTTON Class	
Message	**Description**
BM_CLICK	Sends a WM_BUTTON1DOWN and WM_BUTTON1UP message to itself to simulate a user button selection.
BM_QUERYCHECK	Returns the checked state of the button.
BM_QUERYCHECKINDEX	Returns the 0-based index to the selected button in a group. Returns -1 if no button in the group is selected or if the button receiving the message is not a radio button or an auto-radio button.
BM_QUERYHILITE	Returns the highlighted state of the button.
BM_SETCHECK	Sets the checked state of the button and returns the previous checked state.
BM_SETDEFAULT	Sets the default button state and redraws the button.
BM_SETHILITE	Sets the highlighted state of the button and returns the previous highlighted state.

Message	Description
WM_BUTTON1DBLCLK	Marks button 1, sending a BN_DBLCLICKED notification code when the button-up message arrives.
WM_BUTTON1DOWN	Sets the button 1 window so it can capture mouse input.
WM_BUTTON1UP	If the button 1 window can capture mouse input, and if the mouse pointer is inside button 1 when the button is released, this message releases the mouse and sends a notification message to the owner window. If the button is a push button, the push button control posts a WM_COMMAND message; otherwise, the button control sends a WM_CONTROL message with the BN_CLICKED notification code.
WM_CHAR	Sets the button window so it can capture mouse input when the spacebar is pressed; releases the mouse when the spacebar is released. Passes other key messages to the default window procedure.
WM_CREATE	Validates the requested button style and sets the window text.
WM_DESTROY	Frees the memory containing the window's text.
WM_ENABLE	Sent when an application changes the enabled state of a window.
WM_MATCHMNEMONIC	Returns TRUE if *mp1* matches a mnemonic in the control window's text.
WM_MOUSEMOVE	Sets the default mouse pointer. If the button has the mouse captured, the button's highlighted state changes as the mouse pointer moves in and out of the button boundary.
WM_PAINT	Draws the button according to its style and current state.
WM_QUERYDLGCODE	Returns the DLGC_BUTTON code combined with other DLGC_ codes that designate the button's type.
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.
WM_SETFOCUS	Creates a cursor if the button-control window is receiving the focus. Destroys the cursor if the button-control window is losing the focus.
WM_SETWINDOWPARAMS	Sets the requested window parameters and redraws the button, including the cursor, if the button-control window has the focus.

Table 8-2 (Page 2 of 2). Messages Processed by the WC_BUTTON Class

Button Notification Messages

A button that was created using the BS_PUSHBUTTON or BS_USERBUTTON style posts a WM_COMMAND message to its owner when the user selects it. An application can change this behavior by combining the BS_HELP or BS_SYSCOMMAND styles with the BS_PUSHBUTTON or BS_USERBUTTON styles when creating the button.

A button control that has a style other than BS_PUSHBUTTON or BS_USERBUTTON sends WM_CONTROL messages to its owner when the user selects it.

When the user selects a push button using the mouse pointer, the system automatically highlights the button. The button's window procedure tracks the movement of the pointer until the user releases the button. If the user moves the pointer so that it is outside the button boundary, the system turns off the highlight. The push button control does not post a WM_COMMAND message until the user releases the pointer button, and then, only if the button is released inside the push button boundary. When the owner window receives a WM_COMMAND message from a push button, the low word of the first parameter in the message contains the identifier of the button as specified either in the dialog template or in the WinCreateWindow function when the button was created.

An application should avoid duplicating identifiers for menu items and button controls, because both the items and the controls post identifiers to owner windows as WM_COMMAND messages. However, the application can determine whether a WM_COMMAND message came from a menu or a push button control by looking for the value CMDSRC_MENU or CMDSRC_PUSHBUTTON in the low word of the message's second parameter.

When the user selects any button other than a push button, that button sends a WM_CONTROL message. The application can examine SHORT1FROMMP(*mp1*) in the WM_CONTROL message to find the button identifier, and can examine SHORT2FROMMP(mp2) to determine the notification code for the control message. The notification code can be one of the following:

Table 8-3. Notification Code for Button Control Messages

Code	Description
BN_CLICKED	The user selected the button.
BN_DBLCLICKED	The user double-clicked the button.
BN_PAINT	A user-defined button needs to be drawn. Buttons with the BS_USERBUTTON style send this notification code to instruct the owner window to draw the button control. The second message parameter of the WM_CONTROL message contains a pointer to a USERBUTTON structure that contains the information necessary for drawing the button.

When the user selects a check box or radio button, the button control sends the WM_CONTROL message with the BN_CLICKED notification code to the owner window. In response, the owner window should set the display state of the button by sending the appropriate message back to the button.

An application need not respond to WM_CONTROL messages sent by an auto-check box or an auto-radio button; the system automatically sets the states of these buttons.

Button States

An application can query and set the highlighted and checked states of its buttons by sending messages to them. An application can obtain the handle of a button by calling WinWindowFromID, using the parent window handle and the identifier of the button. In the case of a dialog window, the parent window would be the dialog window, and the identifier would be the button identifier from the dialog template.

Button-control text is stored as window text. An application can set and retrieve this text by using the WinSetWindowText and WinQueryWindowText functions. To set the size, position, and visibility of a button control, an application uses the standard window functions.

Custom Buttons

An application can customize the appearance of a button by using the BS_USERBUTTON style in combination with other button styles. The owner window receives WM_CONTROL messages for these custom buttons whenever they must be drawn, highlighted, or disabled.

When a button must be drawn, the owner window receives a WM_CONTROL message with the high word of the first parameter equal to BN_PAINT. The second parameter is a pointer to a USERBUTTON structure that contains information the application needs to draw the button.

An application uses the **hwnd** member of the USERBUTTON structure in a call to the WinQueryWindowRect function to find the bounding rectangle for the button. The **hps** member is used as a presentation space for any drawing. The **fsState** member contains flags that tell an application how to draw the button: highlighted, unhighlighted, or disabled. The **fsStateOld** member contains flags that describe the current highlighted, unhighlighted, or disabled state of the button.

Using Button Controls

This section explains how to perform the following tasks:

- Create a dialog template for a button resource.
- Create a button for a client window.

An application creates a group by setting the WS_GROUP style bit for the first member of the group.

Using Buttons in a Dialog Window

You can define dialog-window buttons as part of a dialog template in a resource-definition file, as shown in the following Resource Compiler source-code fragment:

```
DLGTEMPLATE IDD_BUTTON
BEGIN
    DIALOG  "", 2, 10, 10, 235, 180, WS_VISIBLE, FCF_DLGBORDER
    BEGIN
        AUTORADIOBUTTON "Radio~1", ID_RADIO1, 15, 80, 45, 12, WS_GROUP
        AUTORADIOBUTTON "Radio~2", ID_RADIO2, 15, 60, 45, 12
        AUTORADIOBUTTON "Radio~3", ID_RADIO3, 15, 40, 45, 12
        AUTORADIOBUTTON "Radio~4", ID_RADIO4, 15, 20, 45, 12

        PUSHBUTTON "Button 1", ID_PUSH1,  20  100, 50, 14, WS_GROUP
        PUSHBUTTON "Button 2", ID_PUSH2,  75, 100, 50, 14, WS_GROUP
        PUSHBUTTON "Button 3", ID_PUSH3, 130, 100, 50, 14, WS_GROUP

        CHECKBOX "Check Box 1",     ID_CHECK1, 150, 65, 65, 12, WS_GROUP
        CHECKBOX "no toggle",       ID_CHECK2, 150, 40, 58, 12, WS_GROUP
        AUTOCHECKBOX "Check Box 3", ID_CHECK3, 150, 20, 65, 12, WS_GROUP

        DEFPUSHBUTTON "OK",         DID_OK,     75, 26, 46, 20, WS_GROUP
    END
END
```

Figure 8-4. Defining Dialog-Window Buttons in a Dialog Template

Each button in a dialog window has an identifier (for example, ID_RADIO1) that allows an application to identify the source of the WM_COMMAND and WM_CONTROL messages. An application can use the identifier as the second argument of the WinWindowFromID function to retrieve the button-window handle.

The dialog template also contains the text for each button. For push buttons, this text is displayed in a rectangular box. If the text is too long to fit in the box, the text is clipped. For radio buttons and check boxes, text is displayed to the right of the button. A user selects the button by clicking either the button or the text itself.

The WS_GROUP style identifies the beginning of each new group of buttons. In the preceding example, the four auto-radio buttons are in the same group, and each of the other buttons is in its own group. The auto-radio buttons in the first group can be selected one at a time only. An application must ensure that only one check box in a group is selected at a time. The order in which items can be selected in the group can wrap around from the end of the item list to its beginning.

Notice that the DEFPUSHBUTTON style in the preceding example has the identifier DID_OK. It is customary to include an *OK* button with this identifier in most dialog windows to provide a uniform user interface. The DEFPUSHBUTTON style draws a thick border around a button and allows a user to select the button by pressing the spacebar.

The dialog-window procedure for a dialog window that contains buttons must respond to WM_COMMAND and WM_CONTROL messages. A common strategy is to use auto-radio buttons and auto-check boxes to let the user set a list of capabilities for a command, and, then, let the user execute the command by choosing an **OK** push button. With this strategy, the dialog-window procedure ignores all WM_CONTROL messages that come from auto-radio buttons and auto-check boxes.

When the dialog-window procedure receives a WM_COMMAND message for the *OK* push button, the procedure should query the auto-radio buttons and auto-check boxes to determine which options have been selected.

Using Buttons in a Client Window

An application can create a button control using an application client window as the owner. The following code fragment shows how an application can use buttons in client windows:

```
#define ID_PBWINDOW 110
HWND hwndButton,hwndClient;

/* Create a button window. */
hwndButton = WinCreateWindow(hwndClient,   /* Parent window  */
    WC_BUTTON,                             /* Class window   */
    "Test Button"                          /* Button text    */
    WS_VISIBLE |                           /* Visible style  */
    BS_PUSHBUTTON,                         /* Button style   */
    10, 10,                                /* x, y           */
    70, 60,                                /* cx, cy         */
    hwndClient,                            /* Owner window   */
    HWND_TOP,                              /* Top of z-order */
    ID_PBWINDOW,                           /* Identifier     */
    NULL,                                  /* Control data   */
    NULL);                                 /* parameters     */
```

Figure 8-5. Creating a Button Control for a Client Window

Once created in the client window, the button control posts a WM_COMMAND message or sends a WM_CONTROL message to the client-window procedure. This window procedure should examine the message identifier to determine which button posted or sent the message.

An application that has client-window buttons can move and size the buttons when the client window receives a WM_SIZE message. An application can move and size a window by using the WinSetWindowPos function. An application can obtain a window handle for a button control by calling the WinWindowFromID function, specifying the handle of the parent window and the window identifier for each button.

Summary

Following are the OS/2 functions, structures, and messages used with button controls:

Table 8-4. Button-Control Functions

Function Name	Description
WinCreateWindow	Creates a new window.
WinQueryWindowText	Copies window text into a buffer.
WinSetWindowText	Sets the window text for the specified window.
WinWindowFromID	Returns the handle of the child window with the specified identify.

Table 8-5. Button-Control Structure

Structure Name	Description
USERBUTTON	User-button structure.

Table 8-6. Messages Received by a Button control

Message	Description
BM_CLICK	Application sends this message to cause the effect of the user clicking a push button.
BM_QUERYCHECK	Returns the zero-based index of a checked radio button.
BM_QUERYCHECKINDEX	Returns the zero-based index of a checked radio button.
BM_QUERYHILITE	Returns the highlighting state of a button control.
BM_SETCHECK	Sets the checked state of a button control.
BM_SETDEFAULT	Sets the default state of a button control.
BM_SETHILITE	Sets the highlight state of a button control.

Table 8-7. Messages Generated by a Button Control

Message	Description
WM_COMMAND	Occurs when a control has a significant event to notify to its owner, or when a keystroke has been translated by an accelerator table.
WM_CONTROL	Occurs when a control has a significant event to notify to its owner.
WM_CONTROLPOINTER	Sent to a control's owner window when the pointer moves over the control window, allowing the owner to set the pointer.
WM_ENABLE	Sets the enable state of a window.
WM_HELP	Occurs when a control procedure does not expect to receive this message and, therefore, takes no action on it, other than to set count to the default value of NULL.
WM_MATCHMNEMONIC	Sent by the dialog box to a control window to determine whether a typed character matches a mnemonic in its window text.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the button control window procedure window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the button control window procedure window parameters.
WM_SYSCOMMAND	Occurs when a control window has a significant event to notify to its owner, or when a keystroke has been translated by an accelerator table into a WM_SYSCOMMAND.

Chapter 9. List-Box Controls

A *list box* is a control window that displays several text items at a time, one or more of which can be selected by the user. This chapter explains how to create and use list-box controls in PM applications.

About List Boxes

An application uses a list box when it requires a list of selectable fields that is too large for the display area or a list of choices that can change dynamically. Each list item contains a text string and a handle. Usually, the text string is displayed in the list-box window; but the handle is available to the application to reference other data associated with each of the items in the list.

A list box always is owned by another window that receives messages from the list box when events occur, such as when a user selects an item from the list box. Typically, the owner is a dialog window (as shown in Figure 9-1) or the client window of an application frame window. The client- or dialog-window procedure defined by the application responds to messages sent from the list box.

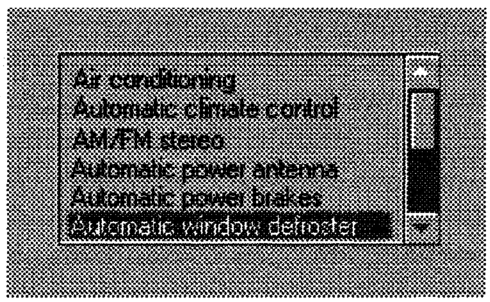


Figure 9-1. List Box in a Dialog Box

A list box always contains a scroll bar for use when the list box contains more items than can be displayed in the list-box window. The list box responds to mouse clicks in the scroll bar by scrolling the list; otherwise, the scroll bar is disabled.

The maximum number of items permitted in a list box is 32767.

Using List Boxes

An application uses a list-box control to display a list in a window. List boxes can be displayed in standard application windows, although they are more commonly used in dialog windows. In either case, notification messages are sent from the list box to its owner window, enabling the application to respond to user actions in the list.

Once a list box is created, the application controls the insertion and deletion of list items. Items can be inserted at the end of the list, automatically sorted into the list, or inserted at a specified index position. Applications can turn list drawing on and off to speed up the process of inserting numerous items into a list.

The owner-window procedure of the list box receives messages when a user manipulates the list-box data. Most default list actions (for example, highlighting

selections and scrolling) are handled automatically by the list box itself. The application controls the responses when the user chooses an item in the list, either by double-clicking the item or by pressing Enter after an item is highlighted. The list box also notifies the application when the user changes the selection or scrolls the list.

Normally, list items are text strings drawn by a list box. An application also can draw and highlight the items in a list. This enables the application to create customized lists that contain graphics. When an application creates a list box with the LS_OWNERDRAW style, the owner of the list box receives a WM_DRAWITEM message for each item that should be drawn or highlighted. This is similar to the owner-drawn style for menus, except that the owner-drawn style applies to the entire list rather than to individual items.

Creating a List-Box Window

List boxes are WC_LISTBOX class windows and are predefined by the system. Applications can create list boxes by calling WinCreateWindow, using WC_LISTBOX as the window-class parameter.

A list box passes notification messages to its owner window, so an application uses its client window, rather than the frame window, as the owner of the list. The client-window procedure receives the messages sent from the list box.

For example, to create a list box that completely fills the client area of a frame window, an application would make the client window the owner and parent of the list-box window, and make the list-box window the same size as the client window. This is shown in the following code fragment:

```
#define ID_LISTWINDOW   250

HWND hwndClient,hwndList;
RECTL rcl;

/* How big is the client window? */
WinQueryWindowRect(hwndClient, &rcl);

/* Make a list-box window.       */
hwndList = WinCreateWindow(hwndClient,    /* Parent       */
    WC_LISTBOX,                           /* Class        */
    "",                                   /* Name         */
    WS_VISIBLE | LS_NOADJUSTPOS,          /* Style        */
    0, 0,                                 /* x, y         */
    rcl.xRight, rcl.yTop,                 /* cx, cy       */
    hwndClient,                           /* Owner        */
    HWND_TOP,                             /* Behind       */
    ID_LISTWINDOW,                        /* ID           */
    NULL,                                 /* Control data */
    NULL);                                /* parameters   */
```

Because the list box draws its own border, and a frame-window border already surrounds the client area of a frame window due to the adjacent frame controls, the effect is a double-thick border around the list box. You can change this effect by calling WinInflateRect to overlap the list-box border with the surrounding frame-window border, resulting in only one list-box border.

Notice that the code specifies the list-box window style LS_NOADJUSTPOS. This ensures that the list box is created exactly the specified size. If the LS_NOADJUSTPOS style is not specified, the list-box height is rounded down, if necessary, to make it a multiple of the item height. Enabling a list box to adjust its height automatically is useful for preventing partial items being displayed at the bottom of a list box.

Using a List Box in a Dialog Window

List boxes most commonly are used in dialog windows. A list box in a dialog box is a control window, like a push button or an entry field. Typically, the application defines a list box as one item in a dialog template in the resource-definition file, as shown in the following resource compiler source-code fragment:

```
DLGTEMPLATE IDD_OPEN
BEGIN
   DIALOG "Open...", IDD_OPEN, 35, 35, 150, 135,
          FS_DLGBORDER, FCF_TITLEBAR
      BEGIN
          LISTBOX         IDD_FILELIST, 15, 15, 90, 90
          PUSHBUTTON      "Drive", IDD_DRIVEBUTTON, 115, 70, 30, 14
          DEFPUSHBUTTON   "Open", IDD_OPENBUTTON, 115, 40, 30, 14
          PUSHBUTTON      "Cancel", IDD_CANCELBUTTON, 115, 15, 30, 14
      END
END
```

Once the dialog resource is defined, the application loads and displays the dialog box as it would normally. The application inserts items into the list when processing the WM_INITDLG message.

A dialog window with a list box usually has an **OK** button. The user can select items in the list, and then indicate a final selection by double-clicking, pressing Enter, or clicking the **OK** button. When the dialog-window procedure receives a message indicating that the user has clicked the **OK** button, it queries the list box to determine the current selection (or selections, if the list allows multiple selections), and then responds as though it had received a WM_CONTROL message with the LN_ENTER notification code.

Adding or Deleting an Item in a List Box

Applications can add or delete an item in a list box by sending an LM_INSERTITEM or LM_DELETEITEM message to the list-box window. Items in a list are specified with a 0-based index (beginning at the top of the list). A new list is created empty; then, the application initializes the list by inserting items.

The application specifies the text and position for each new item. It can specify an *absolute-position* index or one of the following predefined index values:

Table 9-1. List Item Position Index

Value	Meaning
LIT_END	Insert item at end of list.
LIT_SORTASCENDING	Insert item alphabetically ascending into list.
LIT_SORTDESCENDING	Insert item alphabetically descending into list.

The application must send an LM_DELETEITEM message and supply the absolute-position index of the item when deleting items from a list. The LM_DELETEALL message deletes all items in a list.

One way an application can speed up the insertion of list items is to suspend drawing until it has finished inserting items. This is a particularly valuable approach when using a sorted insertion process (when inserting one item can cause rearrangement of the entire list). You can turn off list drawing by calling WinEnableWindowUpdate, specifying FALSE for the *enable* parameter, and then calling WinShowWindow. This forces a total update when insertion is complete. The following code fragment illustrates this concept:

```
HWND hwndFileList;

/* Disable updates while filling the list. */
WinEnableWindowUpdate(hwndFileList, FALSE);
    .
    . /* Send LM_INSERTITEM messages to insert all new items.    */
    .

/* Now cause the window to update and show the new information.  */
WinShowWindow(hwndFileList, TRUE);
```

Notice that this optimization is unnecessary if an application is adding list items while processing a WM_INITDLG message, because the list box is not visible, and the list-box routines are internally optimized.

Responding to a User Selection in a List Box

When a user chooses an item in a list, the primary notification an application receives is a WM_CONTROL message, with the LN_ENTER control code sent to the owner window of the list. Within the window procedure for the owner window, the application responds to the LN_ENTER control code by querying the list box for the current selection (or selections, in the case of an LS_MULTIPLESEL or LS_EXTENDEDSEL list box).

The LN_ENTER control code notifies the application that the user has selected a list item. A WM_CONTROL message with an LN_SELECT control code is sent to the list-box owner whenever a selection in a list changes, such as when a user moves the mouse pointer up and down a list while pressing the mouse button. In this case, items are selected but not yet *chosen*. An application can ignore LN_SELECT control codes when the selection changes, responding only when the item is actually chosen. Or an application can use LN_SELECT to display context-dependent information that changes rapidly with each selection made by the user.

Handling Multiple Selections

When a list box has the style LS_MULTIPLESEL or LS_EXTENDEDSEL, the user can select more than one item at a time. An application must use different strategies when working with these types of lists. For example, when responding to an LN_ENTER control code, it is not sufficient to send a single LM_QUERYSELECTION message, because that message will find only the first selection. To find all current selections, an application must continue sending LM_QUERYSELECTION messages, using the return index of the previous message as the starting index of the next message, until no items are returned.

Creating an Owner-Drawn List Item

To draw its own list items, an application must create a list that has the style LS_OWNERDRAW: the owner window of the list box must respond to the WM_MEASUREITEM and WM_DRAWITEM messages.

When the owner window receives a WM_MEASUREITEM message, it must return the height of the list item. All items in a list must have the same height (greater than or equal to 1). The WM_MEASUREITEM message is sent when the list box is created, and every time an item is added. You can change the item height by sending an LM_SETITEMHEIGHT message to the list-box window.

The owner window receives a WM_DRAWITEM message whenever an item in an owner-drawn list should be drawn or highlighted. Although it is quite common for an owner-drawn list to draw items, it is less common to override the system-default method of highlighting. (This method inverts the rectangle that contains the item.) Do not create your own highlighting unless, for some reason, the system-default method is unacceptable to you.

The WM_DRAWITEM message contains a pointer to an OWNERITEM data structure. The OWNERITEM structure contains the window identifier for the list box, a presentation-space handle, a bounding rectangle for the item, the position index for the item, and the application-defined item handle. This structure also contains two fields that determine whether a message draws, highlights, or removes the highlighting from an item. The OWNERITEM structure has the following form:

```
typedef struct _OWNERITEM { /* oi */
    HWND     hwnd;
    HPS      hps;
    ULONG    fsState;
    ULONG    fsAttribute;
    ULONG    fsStateOld;
    ULONG    fsAttributeOld;
    RECTL    rclItem;
    LONG     idItem;
    ULONG    hItem;
} OWNERITEM;
```

When the item must be drawn, the owner window receives a WM_DRAWITEM message with the **fsState** field set differently from the **fsStateOld** field. If the owner window draws the item in response to this message, it returns TRUE, telling the system not to draw the item. If the owner window returns FALSE, the system draws the item, using the default list-item drawing method.

You can get the text of a list item by sending an LM_QUERYITEMTEXT message to the list-box window. You should draw the item using the *hps* and *rclItem* arguments provided in the OWNERITEM structure.

If the item being drawn is currently selected, the **fsState** and **fsStateOld** fields are both TRUE; they both will be FALSE if the item is not currently selected. The window receiving a WM_DRAWITEM message can use this information to highlight the selected item at the same time it draws the item. If the owner window highlights the item, it must leave the **fsState** and **fsStateOld** fields equal to each other. If the system provides default highlighting for the item (by inverting the item rectangle), the owner window must set the **fsState** field to **1** and the **fsStateOld** field to **0** before returning from the WM_DRAWITEM message.

The owner window also receives a WM_DRAWITEM message when the highlight state of a list item changes. For example, when a user clicks an item, the highlighting must be removed from the currently selected item, and the new selection must be highlighted. If these items are owner-drawn, the owner window receives one WM_DRAWITEM message for each unhighlighted item and one message for the newly highlighted item. To highlight an item, the **fsState** field must equal TRUE, and the **fsStateOld** field must equal FALSE. In this case, the application should highlight the item and return the **fsState** and **fsStateOld** fields equal to FALSE, which tells the system not to highlight the item. The application also can return the **fsState** and **fsStateOld** fields with two different (unequal) values and the list box will highlight the item (the default action).

To remove highlighting from an item, the **fsState** field must equal FALSE and the **fsStateOld** field must equal TRUE. In this case, the application removes the highlighting and returns both the **fsState** and the **fsStateOld** fields equal to FALSE. This tells the system not to attempt to remove the highlighting. The application also can return the **fsState** and **fsStateOld** fields with two different (unequal) values, and the list box will remove the highlighting (the default response).

The following code fragment shows these selection processes:

```
OWNERITEM *poi;

case WM_DRAWITEM:

    /* Convert mp2 into an OWNERITEM structure pointer.                  */
    poi = (POWNERITEM) PVOIDFROMMP(mp2);

    /* Test to see if this is drawing or highlighting/unhighlighting.   */
    if (poi->fsState != poi->fsStateOld) {

        /* This is either highlighting or unhighlighting.               */
        if (poi->fsState) {
            .
            . /* Highlight the item.                                    */
            .
        }
        else {
            .
            . /* Remove the highlighting.                               */
            .
        }

        /* Set fsState = fsStateOld to tell system you did it.          */
        poi->fsState = poi->fsStateOld = 0;

        return TRUE;  /* Tells list box you did the highlighting.       */

    }
    else {
        .
        . /* Draw the item.                                             */
        .
        if (poi->fsState) {   /* Checks to see if item is selected      */
            .
            . /* Highlight the item.                                    */
            .
            /* Set fsState = fsStateOld to tell system you did it.      */
        }
        return TRUE;  /* Tells list box you did the drawing.            */
    }
```

Default List-Box Behavior

This following table lists all the messages handled by the predefined list-box window-class procedure.

Table 9-2 (Page 1 of 2). Messages Handled by WC_LISTBOX Class

Message	Description
LM_DELETEALL	Deletes all items in the list.
LM_DELETEITEM	Removes the specified item from the list, redrawing the list as necessary. Returns the number of items remaining in the list.
LM_INSERTITEM	Inserts a new item in the list according to the position information passed with the message.
LM_QUERYITEMCOUNT	Returns the number of items in the list.
LM_QUERYITEMHANDLE	Returns the specified item handle.
LM_QUERYITEMTEXT	Copies the text of the specified item to a buffer supplied by the message sender.
LM_QUERYITEMTEXTLENGTH	Returns the text length of the specified item.
LM_QUERYSELECTION	For a single-selection list box, returns the zero-based index of the currently selected item. For a multiple-selection list box, returns the next selected item or LIT_NONE if no more items are selected.
LM_QUERYTOPINDEX	Returns the zero-based index to the item currently visible at the top of the list.
LM_SEARCHSTRING	Searches the list for a match to the specified string.
LM_SELECTITEM	Selects the specified item. If the list is a single-selection list, deselects the previous selection. Sends a WM_CONTROL message (with the LN_SELECT code) to the owner window.
LM_SETITEMHANDLE	Sets the specified item handle.
LM_SETITEMHEIGHT	Sets the item height for the list. All items in the list have the same height.
LM_SETITEMTEXT	Sets the text for the specified item.
LM_SETTOPINDEX	Shows the specified item as the top item in the list window, scrolling the list as necessary.
WM_ADJUSTWINDOWPOS	If the list box has the style LS_NOADJUSTPOS, makes no changes to the SWP structure and returns FALSE. Otherwise, adjusts the height of the list box so that a partial item is not shown at the bottom of the list. Returns TRUE if the SWP structure is changed.
WM_BUTTON2DOWN	Returns TRUE; the message is ignored.
WM_BUTTON3DOWN	Returns TRUE; the message is ignored.
WM_CHAR	Processes virtual keys for line and page scrolling. Sends an LN_ENTER notification code for the Enter key. Returns TRUE if the key is processed; otherwise, passes the message to the WinDefWindowProc function.
WM_CREATE	Creates an empty list box with a scroll bar.

Table 9-2 (Page 2 of 2). Messages Handled by WC_LISTBOX Class	
Message	**Description**
WM_DESTROY	Destroys the list and deallocates any memory allocated during its existence.
WM_ENABLE	Enables the scroll bar if there are more items than can be displayed in a list-box window.
WM_MOUSEMOVE	Sets the mouse pointer to the arrow shape and returns TRUE to show that the message was processed.
WM_PAINT	Draws the list box and its items.
WM_SETFOCUS	If the list box is gaining the focus, creates a cursor and sends an LN_SETFOCUS notification code to the owner window. If the list box is losing the focus, this message destroys the cursor and sends an LN_KILLFOCUS notification code to the owner window.
WM_TIMER	Uses timers to control automatic scrolling that occurs when a user drags the mouse pointer outside the window.
WM_SCROLL	Handles scrolling indicated by the list-box scroll bar.

Summary

Following are the operating system structure, functions, and messages used with list boxes.

Table 9-3. List-Box Structure	
Structure Name	**Description**
OWNERITEM	Owner item.

Table 9-4. List-Box Functions	
Function Name	**Description**
WinDeleteLboxItem	Deletes the indexed item from the list box. Returns the number of items left.
WinInsertLboxItem	Inserts text into a list box at index. Returns the actual index where it was inserted.
WinQueryLboxCount	Returns the number of items in the list box.
WinQueryLboxItemText	Fills the buffer with the text of the indexed item. Returns the length of the text.
WinQueryLboxItemTextLength	Returns the length of the text of the indexed item in the list box.
WinQueryLboxSelectedItem	Returns the index of the selected item in the list box. For single selection only.
WinSetLboxItemText	Sets the text of the list box indexed item to buffer.

Table 9-5. Messages Generated by a List Box to Its Owner	
Message	**Description**
WM_CONTROL	Occurs when a list box control has a significant event to notify to its owner.
WM_DRAWITEM	Notification sent to the owner of a list box control each time an item is to be drawn.
WM_MEASUREITEM	Notification sent to the owner of a specific list box control to establish the height and width of an item in that control.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the list box control window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the list box control window parameters.

Table 9-6. Messages Received by a List Box	
Message	**Description**
LM_DELETEALL	Sent to a list box control to delete all the items in the list box.
LM_DELETEITEM	Deletes an item from the list box control.
LM_INSERTITEM	Inserts an item into a list box control
LM_QUERYITEMCOUNT	Returns a count of the number of items in the list box control.
LM_QUERYITEMHANDLE	Returns the handle of the indexed item of the list box control.
LM_QUERYITEMTEXT	Returns the text of the specified list box item.
LM_QUERYITEMTEXTLENGTH	Returns the length of the text of the specified list box item.
LM_QUERYSELECTION	Used to enumerate the selected item, or items, in a list box.
LM_QUERYTOPINDEX	Obtains the index of the item currently at the top of the list box.
LM_SEARCHSTRING	Returns the index of the list box item whose text matches the string.
LM_SELECTITEM	Used to set the selection state of an item in a list box.
LM_SETITEMHANDLE	Sets the handle of the specified list box item.
LM_SETITEMHEIGHT	Sets the height of the items in a list box.
LM_SETITEMTEXT	Sets the text into the specified list box item.
LM_SETTOPINDEX	Used to scroll a particular item to the top of the list box.

Chapter 10. Combination-Box Controls

A *combination box* is two controls in one: an entry field and a list box. This chapter describes how to use *combination-box controls*, also called *combination boxes* and *prompted entry fields*, to let the user choose and edit items from a list in a PM application.

About Combination Boxes

Combination-box controls enable the user to enter data by typing in the entry field or by choosing from a list in the list box. Figure 10-1 is an example of a combination box.

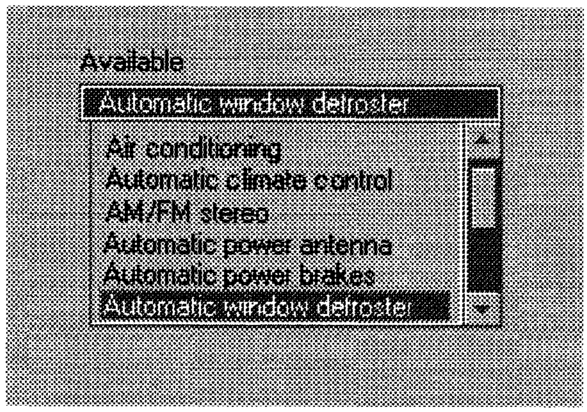


Figure 10-1. Combination Box

A combination-box control automatically manages the interaction between the entry field and the list box. For example, when the user chooses an item in the list box, the combination-box control displays the text for that item in the entry field. Then, the user can edit the text without affecting the item in the list box. When the user types a letter in the entry field, the combination-box control scrolls the list box contents so that items beginning with that letter become visible.

Combination-Box Styles

A combination box can have one of the following styles:

Table 10-1 (Page 1 of 2). Combination-Box Styles	
Style	**Description**
CBS_SIMPLE	Creates a simple combination box, which always displays its list box. The user can enter and edit text in the entry field or choose items from the list box.

Style	Description
CBS_DROPDOWN	Creates a drop-down combination box, which displays its list box only if the user clicks the drop-down icon at the right end of the entry field. See Figure 10-2 for an example of a drop-down combination box. The combination-box control hides the list box when the user clicks the icon a second time. In a drop-down combination box, the user can enter and edit text in the entry field or choose items from the list box.
CBS_DROPDOWNLIST	Creates a drop-down-list combination box, which is similar to the drop-down combination box, except that the user can choose items only from the list box. The user cannot enter or edit text in the entry field. See Figure 10-3 following this table for an example of a drop-down list box.

Table 10-1 (Page 2 of 2). Combination-Box Styles

For combination boxes that have the CBS_DROPDOWN or CBS_DROPDOWNLIST styles, an application can display the list by using the CBM_SHOWLIST message.

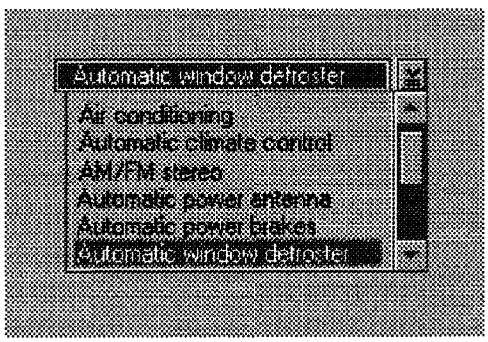


Figure 10-2. Drop-Down Combination Box

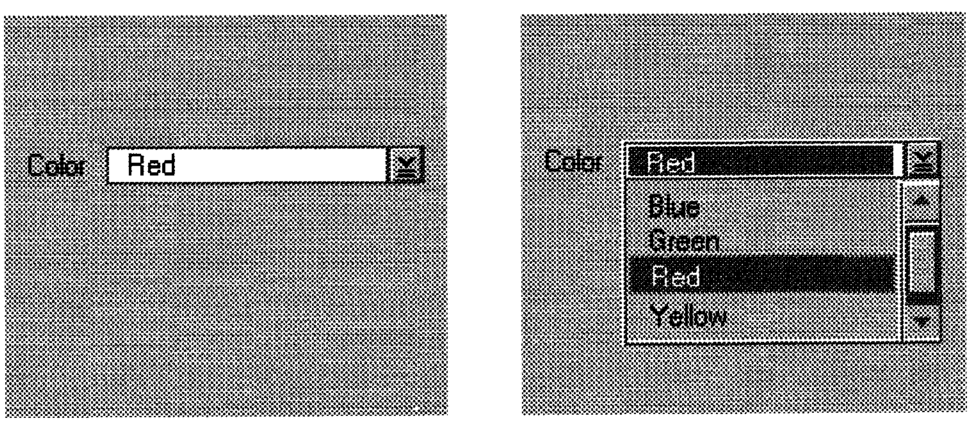


Figure 10-3. Drop-Down List Box

An application can determine whether the list is already showing by using the CBM_ISLISTSHOWING message.

Applications also can use any of the entry-field (EM_) and list-box (LM_) messages with combination boxes. Entry-field messages affect the entry field; list-box messages affect the list box. For example, an application can use the LM_INSERTITEM message to insert items into the list box.

Notification Codes

A combination-box control sends WM_CONTROL messages containing notification codes to its parent window. These notification codes are similar to those sent by entry-field and list-box controls. A combination-box control sends the following notification codes to its owner window:

Table 10-2. Combination-Box Notification Codes

Code	Description
CBN_EFCHANGE	Indicates that the text in a combination-box entry field has changed.
CBN_EFSCROLL	Indicates that the text in a combination-box entry field has been scrolled.
CBN_ENTER	Indicates that a combination-box item has been selected.
CBN_LBSCROLL	Indicates that a combination-box list has been scrolled.
CBN_LBSELECT	Indicates that a combination-box list item has been selected.
CBN_MEMERROR	Indicates that the combination-box control cannot allocate sufficient memory.
CBN_SHOWLIST	Indicates that a combination-box list has dropped down (is visible).

Using Combination Boxes

You can create a combination box by using the WinCreateWindow function or by specifying a COMBOBOX statement in a dialog-window template in a resource file. When creating a combination box using WinCreateWindow, you must specify the predefined class WC_COMBOBOX. If you do not specify a style, the function uses the default styles WS_GROUP, WS_TABSTOP, and WS_VISIBLE.

Summary

The following table lists the OS/2 messages used with combination-box controls:

Table 10-3. Messages Received by a Combination Box

Message	Description
CBM_HILITE	Sets the highlighting state of the entry field control.
CMB_ISLISTSHOWING	Determines whether the list box control is showing.
CBM_SHOWLIST	Sets the showing state of the list box control.

Table 10-4. Message Sent From a Combination Box to Its Owner

Message	Description
WM_CONTROL	Occurs when a control has a significant event to notify to its owner.

Chapter 11. Menus

A *menu* is a window that contains a list of items— text strings, bit maps, or images drawn by the application—that enables the user, by mouse or keyboard, to choose from these predetermined choices. This chapter describes how to use menus in your PM applications.

About Menus

A menu always is owned by another window, usually a frame window. When a user makes a choice from a menu, the menu posts a message containing the unique identifier for the menu item to its owner by way of the owner window's window procedure.

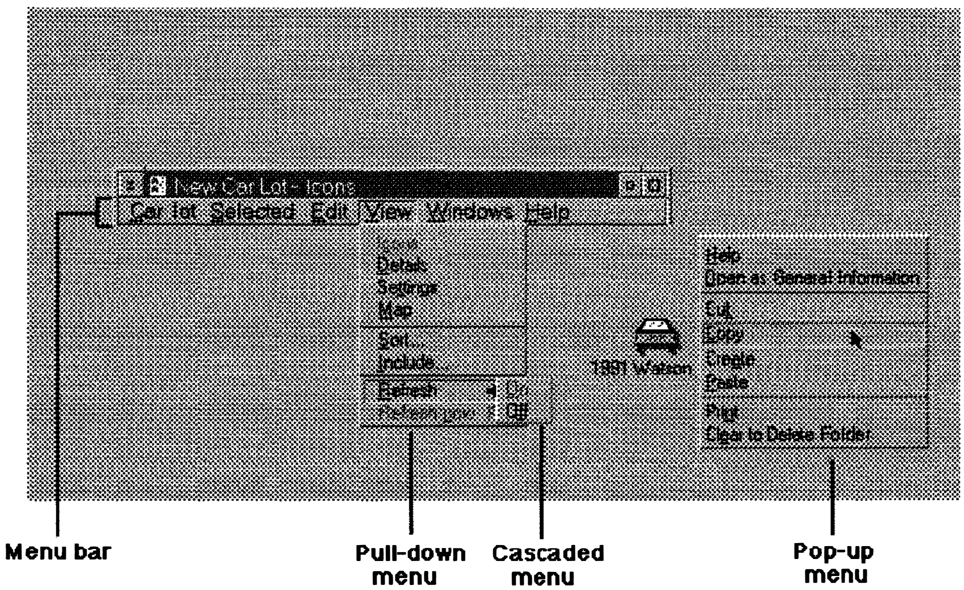


Figure 11-1. Menus

An application typically defines its menus using Resource Compiler, and then associates the menus with a frame window when the frame window is created. Applications also can create menus by filling in menu-template data structures and creating windows with the WC_MENU class. Either way, applications can add, delete, or change menu items dynamically by issuing messages to menu windows.

Menu Bar and Pull-Down Menus

A typical application uses a menu bar and several pull-down submenus. The pull-down submenus ordinarily are hidden, but become visible when the user makes selections in the menu bar. Pull-down submenus always are attached to the menu bar.

The menu bar is a child of the frame window; the menu bar window handle is the key to communicating with the menu bar and its submenus. You can retrieve this handle by calling WinWindowFromID, with the handle of the parent window and the FID_MENU frame-control identifier. Most messages for the menu bar and its

submenus can be issued to the menu-bar window. Flags in the messages tell the window whether to search submenus for requested menu items.

Pop-Up Menus

A pop-up menu is like a pull-down submenu, except that it is not attached to the menu bar; it can appear anywhere in its parent window. A pop-up menu usually is associated with a *portion* of a window, such as the client window (see Figure 11-2); or it is associated with a specific object, such as an icon.

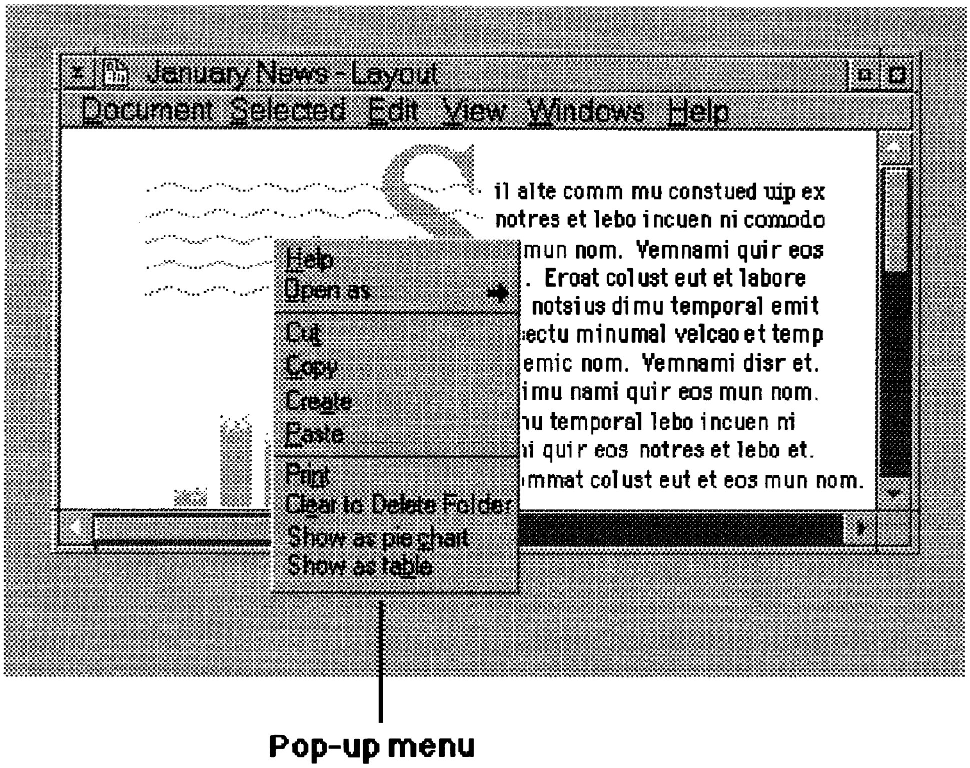


Figure 11-2. Pop-Up Menu

A pop-up menu remains hidden until the user selects it (either by moving the cursor to the appropriate location and pressing Enter or clicking on the location with the mouse). Typically, pop-up menus are displayed at the position of the cursor or mouse pointer; they provide a quick mechanism for selecting often-used menu items.

To include a pop-up menu in an application, you first must define a menu resource in a resource-definition file, then load the resource using the WinLoadMenu or WinCreateMenu functions. You must call WinPopupMenu to create the pop-up menu and display it in the parent window. Applications typically call WinPopupMenu in a window procedure in response to a user-generated message, such as WM_BUTTON2DBLCLK or WM_CHAR.

WinPopupMenu requires that you specify the pop-up menu's handle and also the handles of the parent and owner windows of the pop-up menu. WinLoadMenu and WinCreateMenu return the handle of the pop-up menu window, but you must obtain the handles of the parent and owner by using the WinWindowFromID function.

You determine the position of the pop-up menu in relation to its parent by specifying coordinates and style flags in WinPopupMenu. The *x* and *y* coordinates determine

the position of the lower-left corner of the menu relative to the lower-left corner of the parent. The system may adjust this position, however, if you include the PU_HCONSTRAIN or PU_VCONSTRAIN style flags in the call to WinPopupMenu. If necessary, PU_HCONSTRAIN adjusts the horizontal position of the menu so that its left and right edges are within the borders of the desktop window. PU_VCONSTRAIN makes the same adjustments vertically. Without these flags, a desktop-level pop-up menu can lie partially off the screen, with some items not visible nor selectable.

The PU_POSITIONONITEM flag also can affect the position of the pop-up menu. This flag positions the pop-up menu so that, when the pop-up menu appears, the specified item lies directly under the mouse pointer. Also, PU_POSITIONONITEM automatically selects the item. PU_POSITIONONITEM is useful for placing the current menu selection under the pointer so that, if the user releases the mouse button without selecting a new item, the current selection remains unchanged.

The PU_SELECTITEM flag is similar to PU_POSITIONONITEM except that it just selects the specified item; it does not affect the position of the menu.

You can enable the user to choose an item from a pop-up menu by using the same mouse button that was used to display the menu. To do this, specify the PU_MOUSEBUTTON*n* flag, where *n* corresponds to the mouse button used to display the menu. This flag specifies the mouse buttons for the user to interact with a pop-up menu once it is displayed.

By using the PU_MOUSEBUTTON*n* flag, you can enable the user to display the pop-up menu, select an item, and dismiss the menu, all in one operation. For example, if your window procedure displays the pop-up window when the user double-clicks mouse button 2, specify the PU_MOUSEBUTTON2DOWN flag in the WinPopupMenu function. Then, the user can display the menu with mouse button 2; and, while holding the button down, select an item. When the user releases the button, the item is chosen and the menu dismissed.

System Menu

The system menu in the upper-left corner of a standard frame window is different from the menus defined by the application. The system menu is controlled and defined almost exclusively by the system; your only decision about it is whether to include it when creating a frame window. (It is unusual for a frame window *not* to include a system menu.) The system menu generates WM_SYSCOMMAND messages instead of WM_COMMAND messages. Most applications simply use the default behavior for WM_SYSCOMMAND messages, although applications can add, delete, and change system-menu entries.

Menu Items

All menus can contain two main types of menu items: command items and submenu items. When the user chooses a command item, the menu immediately posts a message to the parent window. When the user selects a submenu item, the menu displays a submenu from which the user may choose another item. Since a submenu window also can contain a submenu item, submenus can originate from other submenus.

When the user chooses a command item from a menu, the menu system posts a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message to the owner window, depending on the style bits of the menu item.

Applications can change the attributes, style, and contents of menu items, and insert and delete items at run time, to reflect changes in the command environment. An application also can add items to or delete items from the menu bar, a pop-up menu, or a submenu. For example, an application might maintain a menu of the fonts currently available in the system. This application would use graphics programming interface (Gpi) calls to determine which fonts were available and, then, insert a menu item for each font into a submenu. Furthermore, the application might set the check-mark attribute of the menu item for the currently chosen font. When the user chose a new font, the application would remove the check-mark attribute from the previous choice and add it to the new choice.

The Help Item

To present a standard interface to the novice user, all applications must have a Help item in their menu bars. The Help item is defined with a particular style, attributes, and position in the menu. When the user chooses the Help item, the menu posts a WM_HELP message to the owner window, enabling the application to respond appropriately.

The item should read `Help`, have an identifier of 0, and have the MIS_BUTTONSEPARATOR or MIS_HELP item styles. The Help menu item should be the last item in the menu template, so that it is displayed as the rightmost item in the menu bar.

If an application uses the system default accelerator table, the user can select the Help item using either a mouse or the F1 key.

Menu-Item Styles

All menu items have a combination of style bits that determine what kind of data the item contains and what kind of message it generates when the user selects it. For example, a menu item can have the MIS_TEXT, MIS_BITMAP, or other styles that specify the visual representation of the menu item on the screen. Other styles determine what kinds of messages the item sends to its owner and whether the owner draws the item. Menu-item styles typically do not change during program execution, but you can query and set them dynamically by sending MM_QUERYITEM and MM_SETITEM messages with the menu-item identifier to the menu-bar window. For text menu items (MIS_TEXT), an MM_SETITEMTEXT message sets the text. The MM_QUERYITEMTEXT message queries the text of the item. For non-text menu items, the **hItem** field of the MENUITEM structure typically contains the handle of a display object, such as a bit-map handle for MIS_BITMAP menu items.

An application can draw a menu item by setting the style MIS_OWNERDRAW for the menu item. This usually is done by specifying the MIS_OWNERDRAW style for the menu item in the resource-definition file; but it also can be done at run time. When the application draws a menu item, it must respond to messages from the menu each time the item must be drawn.

Menu-Item Attributes

Menu items have attributes that determine how the items are displayed and whether or not the user can choose them. An application can set and query menu-item attributes by sending MM_SETITEMATTR and MM_QUERYITEMATTR messages, with the menu-item identifier, to the menu-bar window. If the specified item is in a submenu, there are two methods of determining its attributes. The first is to send MM_SETITEMATTR and MM_QUERYITEMATTR messages to the top-level menu, specifying the identifier of the item and setting a flag so that the message searches all submenus for the item. Then, you can retrieve the handle of the menu-bar by

calling WinWindowFromID, with the handle of the frame window and the FID_MENU frame-control identifier.

The second method, which is more efficient if you want to either work with more than one submenu item or set the same item several times, involves two steps:

1. Send an MM_QUERYITEM message to the menu, with the identifier of the submenu. The updated MENUITEM structure contains the window handle of the submenu.
2. Send an MM_QUERYITEMATTR (or MM_SETITEMATTR) message to the submenu window, specifying the identifier of the item in the submenu.

Menu-Item Structure

A single menu item is defined by the MENUITEM data structure. This structure is used with the MM_INSERTITEM message to insert items in a menu or to query and set item characteristics with the MM_QUERYITEM and MM_SETITEM messages. The MENUITEM structure has the following form:

```
typedef struct _MENUITEM { /* mi */
    SHORT  iPosition;
    USHORT afStyle;
    USHORT afAttribute;
    USHORT id;
    HWND   hwndSubMenu;
    ULONG  hItem;
} MENUITEM;
```

You can derive the values of most of the fields in this structure directly from the resource-definition file. However, the last field in the structure, **hItem**, depends on the style of the menu item.

The **iPosition** field specifies the ordinal position of the item within its menu window. If the item is part of the menu bar, **iPosition** specifies its relative left-to-right position, with 0 being the leftmost item. If the item is part of a submenu, **iPosition** specifies its relative top-to-bottom and left-to-right positions, with 0 being the upper-left item. An item with the MIS_BREAKSEPARATOR style in a pull-down menu causes a new column to begin.

The **afStyle** field contains the style bits of the item. The **afAttribute** field contains the attribute bits.

The **id** field contains the menu-item identifier. The identifier *should* be unique but does not *have* to be. Just remember that, when multiple items have the same identifier, they post the same command number in the WM_COMMAND, WM_SYSCOMMAND, and WM_HELP messages. Also, any message that specifies a menu item with a non-unique identifier will find the first item that has that identifier.

The **hwndSubMenu** field contains the window handle of a submenu window (if the item is a submenu item). The **hwndSubMenu** field is NULL for command items.

The **hItem** field contains a handle to the display object for the item, unless the item has the MIS_TEXT style, in which case, **hItem** is 0. For example, a menu item with the MIS_BITMAP style has an **hItem** field that is equal to its bit-map handle.

Menu Access

The OS/2 operating system is designed to work with or without a mouse or other pointing device. The system provides default behavior that enables a user to interact with menus without a mouse. Following are the keystrokes that produce this default behavior:

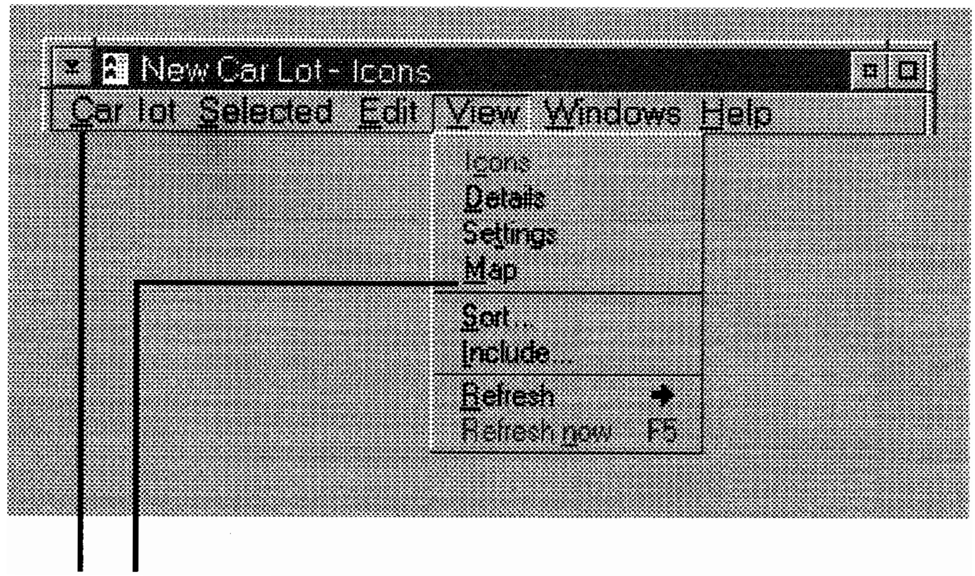
Table 11-1. Keystroke Menu Access

Keystroke	Action
Alt	Toggles in and out of menu-bar mode.
Alt+Spacebar	Shows the system menu.
F10	Backs up one level. If a submenu is displayed, it is canceled. If no submenu is displayed, this keystroke exits the menu.
Shift+Esc	Shows the system menu.
Right Arrow	Cycles to the next top-level menu item. If the selected item is at the far-left side of the menu, the menu code sends a WM_NEXTMENU message to the frame window. The default processing by the frame window is to cycle between the application and system menus. (An application can modify this behavior by subclassing the frame window.) If the selected item is in a submenu, the next column in the submenu is selected, or the next top-level menu item is selected; this keystroke also can send or process a WM_NEXTMENU message.
Left Arrow	Works like the Right Arrow key, except in the opposite direction. In submenus, this keystroke backs up one column, except when the currently selected item is in the far-left column, in which case the previous submenu is selected.
Up Arrow or **Down Arrow**	When pressed in a top-level menu, activates a submenu. When pressed in a submenu, this keystroke selects the previous or next or item, respectively.
Enter	Activates a submenu, and highlights the first item if an item has a submenu associated with it; otherwise, this keystroke chooses the item as though the user released the mouse button while the item was selected.
Alphabetic character	Selects the first menu item with the specified character as its mnemonic key. A mnemonic is defined for a menu item by placing a tilde (~) before the character in the menu text. If the selected item has a submenu associated with it, the menu is displayed, and the first item is highlighted; otherwise, the item is chosen.

An application does not support the default keyboard behavior with any unusual code; instead, the application receives a message when a menu item is chosen by the keyboard just as though it had been chosen by a mouse.

Mnemonics

Adding mnemonics to menu items is one way of providing the user with keyboard access to menus. You can indicate a mnemonic keystroke for a menu item by preceding a character in the item text with a tilde, as in *~nFile*; Figure 11-3 on page 11-7 shows the result on screen. Then, the user can choose that item by pressing the mnemonic key when the menu is active.



Mnemonics

Figure 11-3. Examples of Mnemonics

The menu bar is *active* when the user presses and releases the Alt key, and the first item in the menu bar is highlighted. A pop-up or pull-down menu is active when it is *open*.

Accelerators

In addition to mnemonics, a menu item can have an associated *keyboard accelerator*. Accelerators are different from mnemonics in that the menu need not be active for the accelerator key to work. If you have associated a menu item with a keyboard accelerator, display the accelerator to the right of the menu item. Do this in the resource-definition file by placing a tab character (\t) in the menu text before the characters that will be displayed on the right. For example, if the Close item had the F3 function key as its keyboard accelerator, the text for the item would be `Close\tF3`.

Using Menus

This section explains how to perform the following tasks:

- Define menu items in a resource file.
- Include a menu bar in a standard window.
- Create a pop-up menu.
- Add a menu to a dialog window.
- Access the system menu.
- Respond to a the menu choice of a user.
- Set and query menu-item attributes.
- Add and delete menu items.
- Create a custom menu item.

Defining Menu Items in a Resource File

Typically, a menu resource represents the menu bar or pop-up menu and all the related submenus. A menu-item definition is organized as shown in the following code:

```
MENUITEM item text, item identifier, item style, item attributes
```

The menu resource-definition file specifies the text of each item in the menu, its unique identifier, its style and attributes, and whether it is a command item or a submenu item. A menu item that has no specification for style or attributes has the default style of MIS_TEXT and all attribute bits off, indicating that the item is enabled. The MIS_SEPARATOR style identifies nonselectable lines between menu items. Following is sample Resource Compiler source code that defines a menu resource. The code defines a menu with three submenu items in the menu bar (**File**, **Edit**, and **Font**) and a command item (**Help**). Each submenu has several command items, and the **Font** submenu has two other submenus within it.

```
MENU ID_MENU_RESOURCE
BEGIN
    SUBMENU "~File", IDM_FILE
        BEGIN
            MENUITEM "~Open...",       IDM_FI_OPEN
            MENUITEM "~Close\tF3",     IDM_FI_CLOSE, 0, MIA_DISABLED
            MENUITEM "~Quit",          IDM_FI_QUIT
            MENUITEM "",               IDM_FI_SEP1, MIS_SEPARATOR
            MENUITEM "~About Sample",  IDM_FI_ABOUT
        END
    SUBMENU "~Edit", IDM_EDIT
        BEGIN
            MENUITEM "~Undo",          IDM_ED_UNDO, 0, MIA_DISABLED
            MENUITEM "",               IDM_ED_SEP1, MIS_SEPARATOR
            MENUITEM "~Cut",           IDM_ED_CUT
            MENUITEM "C~opy",          IDM_ED_COPY
            MENUITEM "~Paste",         IDM_ED_PASTE
            MENUITEM "C~lear",         IDM_ED_CLEAR
        END
    SUBMENU "Font", IDM_FONT
        BEGIN
            SUBMENU "Style",           IDM_FONT_STYLE
                BEGIN
                    MENUITEM "Plain",  IDM_FONT_STYLE_PLAIN
                    MENUITEM "Bold",   IDM_FONT_STYLE_BOLD
                    MENUITEM "Italic", IDM_FONT_STYLE_ITALIC
                END
            SUBMENU "Size",            IDM_FONT_SIZE
                BEGIN
                    MENUITEM "10",     IDM_FONT_SIZE_10
                    MENUITEM "12",     IDM_FONT_SIZE_12
                    MENUITEM "14",     IDM_FONT_SIZE_14
                END
        END
    MENUITEM "F1=Help", 0x00, MIS_TEXT | MIS_BUTTONSEPARATOR | MIS_HELP
END
```

To define a menu item with the MIS_BITMAP style, an application must use a tool such as Icon Editor to create a bit map, include the bit map in its resource-definition file, and define a menu in the file (as shown in the following code fragment). The text for the bit map menu items is an ASCII representation of the resource identifier of the bit map resource to be displayed for that item.

```
/* Bring externally created bit maps into the resource file. */
BITMAP 101 button.bmp
BITMAP 102 hirest.bmp
BITMAP 103 hizoom.bmp
BITMAP 104 hired.bmp

/* Connect a menu item with a bit map.                       */
SUBMENU "~Bitmaps", IDM_BITMAP
    BEGIN
        MENUITEM "#101", IDM_BM_01, MIS_BITMAP
        MENUITEM "#102", IDM_BM_02, MIS_BITMAP
        MENUITEM "#103", IDM_BM_03, MIS_BITMAP
        MENUITEM "#104", IDM_BM_04, MIS_BITMAP
    END
```

Including a Menu Bar in a Standard Window

If you have defined a menu resource in a resource-definition file, you can use the menu resource to create a menu bar in a standard window. You include the menu bar by using the FCF_MENU attribute flag and specifying the menu-resource identifier in a call to WinCreateStdWindow, as shown in the following code fragment:

```
#define ID_MENU_RESOURCE 100

HWND hwndFrame;
CHAR szClassName[]="MyClass";
CHAR szTitle[]="My Title";

ULONG flControlStyle = FCF_MENU     | FCF_SIZEBORDER |
                       FCF_TITLEBAR | FCF_ACCELTABLE;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
    WS_VISIBLE,
    &flControlStyle,
    szClassName,
    szTitle,
    0, (HMODULE) NULL,
    ID_MENU_RESOURCE,
    NULL);
```

After you make this call, the operating system automatically includes the menu in the window, drawing the menu bar across the top of the window. When the user chooses an item from the menu, the menu posts the message to the frame window. The frame window passes any WM_COMMAND messages to the client window. (The frame window does not pass WM_SYSCOMMAND messages to the client window.) WM_HELP messages are posted to the focus window. The WinDefWindowProc function passes WM_HELP messages to the parent window. If a WM_HELP message is passed to a frame window, the frame window calls the

HK_HELP hook. Your client window procedure must process these messages to respond to the user's actions.

Creating a Pop-up Menu

The following code fragment shows how to make a pop-up menu appear when the user double-clicks mouse button 2 anywhere in the parent window. The menu is positioned with the mouse pointer located on the item having the IDM_OPEN identifier and is constrained horizontally and vertically. Then, the user can select an item from the pop-up menu using mouse button 2.

```
#define ID_MENU_RESOURCE  110
#define IDM_OPEN          120

HWND hwndFrame;

MRESULT ClientWndProc(
HWND hwnd,
ULONG msg,
MPARAM mp1,
MPARAM mp2)
{
    HWND hwndMenu;
    BOOL fSuccess;

    switch (msg) {
        .
        .   /* Process other messages. */
        .
        case WM_BUTTON2DBLCLK:
            hwndMenu = WinLoadMenu(hwnd, (HMODULE) NULL, ID_MENU_RESOURCE);
            fSuccess = WinPopupMenu(hwnd,
                                    hwndFrame,
                                    hwndMenu,
                                    20,
                                    50,
                                    IDM_OPEN,
                                    PU_POSITIONONITEM   |
                                    PU_HCONSTRAIN       |
                                    PU_VCONSTRAIN       |
                                    PU_MOUSEBUTTON2DOWN |
                                    PU_MOUSEBUTTON2);

                  .
                  .
                  .
```

Adding a Menu to a Dialog Window

You might want to use menus in windows that were not created using the WinCreateStdWindow function. For these windows, you can load a menu resource by using the WinLoadMenu function and specifying the parent window for the menu. WinLoadMenu assigns the specified menu resource to the parent. To see the menu in the window, you must send a WM_UPDATEFRAME message to the parent after loading the menu resource. This strategy is especially useful for adding menus to a window created as a dialog window, but it can be used no matter what type of window is specified as the parent.

Accessing the System Menu

Although most applications do not alter the system menu, you can obtain the handle of the system menu by calling WinWindowFromID with a frame-window handle (or dialog-window handle) and the identifier FID_SYSMENU. Once you have the handle of the system menu, you can access the individual menu items by using predefined constants. For example, the following code fragment shows how to disable the **Close** menu item in the system menu of a window:

```
HWND hwndSysMenu;
HWND hwndFrame;

hwndSysMenu = WinWindowFromID(hwndFrame, FID_SYSMENU);

WinSendMsg(hwndSysMenu, MM_SETITEMATTR,
    MPFROM2SHORT(SC_CLOSE, TRUE),
    MPFROM2SHORT(MIA_DISABLED, MIA_DISABLED));
```

Responding to a User's Menu Choice

When a user chooses a menu item, the client window procedure receives a WM_COMMAND message with SHORT1FROMMP(mp1) equal to the menu identifier of the chosen item. Your application must use the menu identifier to guide its response to the choice. Typically, the code in the client window procedure resembles the following code fragment:

```
    case WM_COMMAND:
        DoMenuCommand(hwnd, SHORT1FROMMP(mp1));
        return 0;
```

The function that translates the menu identifier into an action typically resembles the following code fragment:

```
VOID DoMenuCommand(
HWND hwnd,
USHORT usItemID)
{

    /* Test the menu item. */
    switch (usItemID) {
        case IDM_FI_NEW:
            DoNew(hwnd);
            break;

      .
      . /* etc. */
      .

    }
}
```

The menu window sends a WM_MENUSELECT message every time the menu selection changes. SHORT1FROMMP(mp1) contains the identifier of the item that is

changing state, and SHORT2FROMMP(mp2) is a 16-bit Boolean value that describes whether or not the item is chosen; the *mp2* parameter contains the handle of the menu.

If the Boolean value is FALSE, the item is selected but not chosen; for example, the user may have moved the cursor or mouse pointer over the item while the button was down. An application can use this message to display Help information at the bottom of the application window. The return value is ignored.

If the Boolean value is TRUE, the item is chosen—that is, the user pressed Enter or released the mouse button while an item was selected. If the application returns FALSE, the menu does not generate a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message, and the menu is not dismissed.

Setting and Querying Menu-Item Attributes

Menu-item attributes are represented in the **fAttribute** field of the MENUITEM data structure. Typically, attributes are set in the resource-definition file of the menu and are changed at run time as required. Applications can use the MM_SETITEMATTR and MM_QUERYITEMATTR messages to set and query attributes for a particular menu item. One of the most common uses of these messages is to check and uncheck menu items to let the user know what option is selected currently. For example, if you have a menu item that should toggle between checked and unchecked each time the user selects it, you can use the following code fragment to change the checked attribute. In this example, you send an MM_QUERYITEMATTR message to the menu item to obtain its current checked attribute; then, you use the exclusive OR operator to toggle the state; and finally, you send the new attribute state back to the item using an MM_SETITEMATTR message.

```
usAttrib = SHORT1FROMMR(
    WinSendMsg(hwndMenu,            /* Submenu window       */
    MM_QUERYITEMATTR,               /* Message              */
    (MPARAM)itemID,                 /* Item identifier      */
    (MPARAM)MIA_CHECKED             /* Attribute mask       */
    ));

usAttrib ^= MIA_CHECKED;        /* XOR to toggle checked attribute */

WinSendMsg(hwndMenu,                       /* Submenu window       */
    MM_SETITEMATTR,                        /* Message              */
    (MPARAM)itemID,                        /* Item identifier      */
    MPFROM2SHORT(MIA_CHECKED, usAttrib)); /* Attribute mask, value */
```

Adding and Deleting Menu Items

An application can add and delete items from its menus dynamically by sending MM_INSERTITEM and MM_DELETEITEM messages to the menu window. Any item, including those in submenus, can be deleted by sending a message to the menu window. Messages to insert items in submenus must be sent to the submenu's window (rather than to the window of the top-level menu). You can retrieve the handle of a submenu of the menu bar by sending an MM_QUERYITEM message to the menu-bar and specifying the identifier of the submenu item for the submenu, as shown in the following code fragment:

```
/* IDM_MYMENUID is the identifier of the submenu containing the item. */

MENUITEM mi;
HWND hwndMenu, hwndSubMenu, hwndPullDown,hwndFrame;

hwndMenu = WinWindowFromID(hwndFrame, FID_MENU);
WinSendMsg(hwndMenu,                              /* Handle of menu bar  */
    MM_QUERYITEM,                                 /* Message             */
    MPFROM2SHORT(IDM_MYMENUID, TRUE),             /* Submenu identifier  */
    (MPARAM) &mi);                                /* Pointer to MENUITEM */

hwndPullDown = mi.hwndSubMenu;                    /* Handle to submenu   */
```

Once the application has the handle of the submenu, it can insert an item by filling in a MENUITEM structure and sending an MM_INSERTITEM message to the submenu. For text-menu items, the application must send a pointer to the text string as well as to the MENUITEM structure.

```
PSZ  pszNewItemString;

mi.iPosition = MIT_END;
mi.afStyle = MIS_TEXT;
mi.afAttribute = 0;
mi.id = IDM_MYMENU_FIRST;
mi.hwndSubMenu = NULL;
mi.hItem = 0;

WinSendMsg(hwndPullDown, MM_INSERTITEM, (MPARAM) &mi,
    (MPARAM) pszNewItemString);
```

To delete an item, the application sends an MM_DELETEITEM message to the menu bar, specifying the identifier of the item to delete. For example, to clear all the items following IDM_MYMENU_FIRST in a submenu in which the items are numbered sequentially, use the following code:

```
USHORT usItemNum;

/* Clear all the items in MYMENU.                 */
hwndMenu = WinWindowFromID(hwndFrame, FID_MENU);
usItemNum = IDM_MYMENU_FIRST;
while (WinSendMsg(hwndMenu, MM_DELETEITEM,
    MPFROM2SHORT(usItemNum++, TRUE), NULL) != 0);
```

Adding a complete submenu to the menu bar is a more complicated procedure than that shown in the previous examples. There are two strategies. The recommended technique is to define all possible submenus in your resource-definition file; and then, as your application runs, selectively remove and insert the submenus as needed.

For example, assume that your application has a submenu that you want to be displayed only when a particular application tool is in use. You must first define the submenu as part of the main menu resource in your resource-definition file, so that

the system reads in the resource menu template and creates the submenu window along with the rest of the menu. You then can remove the submenu from the menu bar, saving the title of the submenu and the MENUITEM structure that defines the submenu, as shown in the following code fragment:

```
HWND hwndMenu, hwndClient;
MENUITEM mi;
CHAR szMenuTitle[MAX_STRINGSIZE];

/* Remove a submenu so that you can replace it later.          */

/* Obtain the handle of a menu.                                */
hwndMenu = WinWindowFromID(WinQueryWindow(hwndClient, QW_PARENT),
                           FID_MENU);

/* Obtain information on the item to remove.                   */
WinSendMsg(hwndMenu, MM_QUERYITEM,
    MPFROM2SHORT(IDM_MENUID, TRUE),  /* TRUE to search submenus */
    (MPARAM)&mi);

/* Save the text for the submenu item.                         */
WinSendMsg(hwndMenu, MM_QUERYITEMTEXT,
    MPFROM2SHORT(IDM_FONT, MAX_STRINGSIZE),
    (MPARAM)szMenuTitle);

/* Remove the item, but retain mi and szMenuTitle.             */
WinSendMsg(hwndMenu, MM_REMOVEITEM,
    MPFROM2SHORT(IDM_FONT, TRUE), NULL);
```

It is important to use the MM_REMOVEITEM message, rather than MM_DELETEITEM, to remove the item; deleting the item destroys the submenu window—removing it does not. The submenu should remain intact so that you can insert it later.

To reinsert the submenu, send an MM_INSERTITEM message to the menu bar, passing the MENUITEM structure and menu title that you saved when you removed the item. The following code fragment shows how to insert a submenu that was removed by using the previous code example:

```
/* Put the submenu back in and obtain the handle of the menu bar. */
hwndMenu = WinWindowFromID(
               WinQueryWindow(hwndClient, QW_PARENT), FID_MENU);

/* Use the information that you saved when you removed the menu.  */
WinSendMsg(hwndMenu, MM_INSERTITEM, (MPARAM)&mi,
           (MPARAM)szMenuTitle);
```

The other technique that you can use to insert a submenu in the menu bar is to build up, in memory, a data structure as a menu template and use that template and WinCreateWindow to create a submenu. The resultant submenu window handle then is placed in the **hwndSubMenu** field of a MENUITEM structure, and the menu item is sent to the menu bar with an MM_INSERTITEM message.

You also can create an empty submenu window by using WinCreateWindow. Pass NULL for the *pCtlData* and *pPresParams* parameters, instead of building the menu

template in memory. Then insert a new menu item in the menu bar by using the MM_INSERTITEM message, setting the MIS_SUBMENU style, and putting the window handle of the created menu into the **hwndSubMenu** field. Then use the MM_INSERTITEM message to insert the items in the new pull-down menu.

Creating a Custom Menu Item

Applications can customize the appearance of an individual menu item by setting the MIS_OWNERDRAW style bit for the item. The operating system sends two different messages to an application that include owner-drawn menu items: WM_MEASUREITEM and WM_DRAWITEM. Both messages include a pointer to an OWNERITEM data structure.

WM_MEASUREITEM is sent only once for each owner-drawn item when the menu is initialized. The message is sent to the owner of the menu (typically, a frame window), which forwards the message to its client window. Typically, the client window procedure processes WM_MEASUREITEM by filling in the **yTop** and **Right** fields of the RECTL structure, specified by the **rclItem** field of this OWNERITEM structure; this specifies the size of the rectangle needed to enclose the item when it is drawn. The following code fragment responds to a WM_MEASUREITEM message.

```
case WM_MEASUREITEM:
    ((POWNERITEM) mp2)->rclItem.xRight = 26;
    ((POWNERITEM) mp2)->rclItem.yTop = 10;
    return 0;
```

If a menu item has the MIS_OWNERDRAW style, the owner window receives a WM_DRAWITEM message every time the menu item needs to be drawn. You process this message by using the **hps** and **rclItem** fields of the OWNERITEM structure to draw the item. There are two situations in which the owner window receives a WM_DRAWITEM message:

- When the item must be redrawn completely
- When the item must be highlighted or have its highlight removed.

You can choose to handle one or both of these situations. Typically, you handle the drawing of the item. You may not want to handle the second situation, however, since the system-default behavior (inverting the bits in the item rectangle) often is acceptable.

The two situations in which a WM_DRAWITEM message is received are detected by comparing the values of the **fsState** and **fsStateOld** fields of the OWNERITEM structure that is sent as part of the message. If the two fields are the same, draw the item. Before drawing the item, however, check its attributes to see whether it has the attributes MIA_CHECKED, MIA_FRAMED, or MIA_DISABLED. Then draw the item according to the attributes.

For example, when the checked attribute of an owner-drawn menu item changes, the system sends a WM_DRAWITEM message to the item so that it can redraw itself and either draw or remove the check mark. If you want the system-default check mark, simply draw the item and leave the **fsAttribute** and **fsAttributeOld** fields unchanged; the system draws the check mark if necessary. If you draw the check mark yourself, clear the MIA_CHECKED bit in both **fsAttribute** and **fsAttributeOld** so that the system does not attempt to draw a check mark.

In the same example, if **fsAttribute** and **fsAttributeOld** are not equal, the highlight showing that an item is selected needs to change. The MIA_HILITED bit of the **fsAttribute** field is set if the item needs to be highlighted and is not set if the highlight needs to be removed. If you do not want to provide your own highlighting, you should ignore any WM_DRAWITEM message in which **fsAttribute** and **fsAttributeOld** are not equal. If you do not alter these two fields, the system performs its default highlighting operation. If you want to provide your own visual cue that an item is selected, respond to a WM_DRAWITEM message in which the **fsAttribute** and **fsAttributeOld** fields are not equal by providing the cue and clearing the MIA_HILITED bit of both fields before returning from the message.

Likewise, the MIA_CHECKED and MIA_FRAMED bits of **fsAttribute** and **fsAttributeOld** either can be used to perform the corresponding action or passed on, unchanged, so that the system performs the action.

The following code fragment shows how to respond to a WM_DRAWITEM message when you want to draw the item and also be responsible for its highlighted state:

```
case WM_DRAWITEM:
    {
    POWNERITEM poi;
    RECTL      rcl;
    MPARAM     mp2;

    poi = (POWNERITEM) mp2;

    /*
     * If the new attribute equals the old attribute,
     * redraw the entire item.
     */

    if (poi->fsAttribute == poi->fsAttributeOld) {

        /*
         * Draw the item in poi->hps and poi->rclItem, and check the
         * attributes for check marks. If you produce your own check marks,
         * use this line of code:
         *
         *     poi->fsAttributeOld = (poi->fsAttribute &= ~MIA_CHECKED;
         */

    }

    /* Else highlight the item or remove its highlight. */

    else if ((poi->fsAttribute & MIA_HILITED) !=
            (poi->fsAttributeOld & MIA_HILITED)) {

        /*
         * Set bits the same so that the menu window does not highlight
         * the item or remove its highlight.
         */

        poi->fsAttributeOld = (poi->fsAttribute &= ~MIA_HILITED);
    }
    return TRUE; /* TRUE means the item is drawn. */
    } /* endcase */
```

Summary

This section lists the OS/2 functions, structures, and messages used with menus.

Table 11-2. Menu Functions

Function Name	Description
WinCreateMenu	Creates a menu window from the menu template.
WinCheckMenuItem	Sets the check state of the specified menu item to the flag.
WinEnableMenuItem	Sets the state of the specified menu item to the enable flag.
WinIsMenuItemChecked	Returns the state (checked/not checked) of the identified menu item.
WinIsMenuItemEnabled	Returns the state (enable/disable) of the specified menu item.
WinIsMenuItemValid	Returns TRUE if the specified item is a valid choice.
WinLoadMenu	Creates a menu window from the menu template **MenuId** from **Resource**, and returns in **Menu** the window handle for the created window.
WinPopupMenu	Displays a pop-up menu.
WinSetMenuItemText	Sets the text for menu indexed item to buffer.

Table 11-3. Menu Structures

Structure Name	Description
MENUITEM	Menu item.
OWNERITEM	Owner item.

Table 11-4 (Page 1 of 2). Messages Received by a Menu

Message	Description
MM_DELETEITEM	Deletes a menu item.
MM_ENDMENUMODE	Sent to a menu control to terminate menu selection.
MM_INSERTITEM	Inserts a menu item in a menu.
MM_ISITEMVALID	Returns the selectable status of a specified menu item.
MM_ITEMIDFROMPOSITION	Returns the identity of a menu item of a specified index.
MM_ITEMPOSITIONFROMID	Returns the index of a menu item of a particular identify.
MM_QUERYITEM	Returns the definition of the specified menu item.
MM_QUERYITEMATTR	Returns the attributes of a menu item.
MM_QUERYITEMCOUNT	Returns the number of items in the menu.
MM_QUERYITEMRECT	Returns the bounding rectangle of a menu item.
MM_QUERYITEMTEXT	Returns the text of the specified menu item.
MM_QUERYITEMTEXTLENGTH	Returns the text length of the specified menu item.

Table 11-4 (Page 2 of 2). Messages Received by a Menu	
Message	**Description**
MM_QUERYSELITEMID	Returns the identity of the selected menu item.
MM_REMOVEITEM	Removes a menu item.
MM_SELECTITEM	Selects or deselects a menu item.
MM_SETITEM	Sets the definition of a menu item.
MM_SETITEMATTR	Sets the attributes of a menu item.
MM_SETITEMHANDLE	Sets the handle of a menu item.
MM_SETITEMTEXT	Sets the text of a menu item.
MM_STARTMENUMODE	Used to begin menu selection.

Table 11-5 (Page 1 of 2). Messages Generated by a Menu	
Message	**Description**
WM_ADJUSTWINDOWPOS	Sent by WinSetWindowPos to enable the window to adjust its new position or size whenever it is about to be moved.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_BUTTON2DOWN	Occurs when the user presses pointer button 2.
WM_BUTTON3DOWN	Occurs when the user presses pointer button 3.
WM_COMMAND	Occurs when a control has a significant event to notify to its owner or when a keystroke has been translated by an accelerator table.
WM_CONTEXTMENU	Occurs when the operator requests a pop-up menu.
WM_CONTROLPOINTER	Sent to the owner window of a control when the pointing device pointer moves over the control window, enabling the owner to set the pointer.
WM_CREATE	Occurs when an application requests the creation of a window.
WM_DESTROY	Occurs when an application requests the destruction of a window.
WM_DRAWITEM	Sent to the owner of a menu control each time an item is to be drawn.
WM_ENABLE	Sets the enable state of a window.
WM_FOCUSCHANGE	Occurs when the window possessing the focus is changed.
WM_HELP	Occurs when a control has a significant event to notify to its owner or when a keystroke has been translated by an accelerator table into a WM_HELP.
WM_INITMENU	Occurs when a menu control is about to become active.
WM_MEASUREITEM	Sent to the owner of a meu control to establish the height for an item in that control.
WM_MENUEND	Occurs when a menu control is about to terminate.
WM_MENUSELECT	Occurs when a menu item has been selected.
WM_MOUSEMOVE	Occurs when the pointing device pointer moves.

Table 11-5 (Page 2 of 2). Messages Generated by a Menu	
Message	**Description**
WM_NEXTMENU	Occurs when either the beginning or the end of the menu is reached using the cursor control keys.
WM_PAINT	Occurs when a window needs repainting.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_SETFOCUS	Occurs when a window is to receive or lose the input focus.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the menu parameters.
WM_SYSCOMMAND	Occurs when a control has a significant event to notify to its owner or when a keystroke has been translated by an accelerator table into a WM_SYSCOMMAND.

Chapter 12. Entry-Field Controls

An *entry field* is a control window that enables a user to view and edit a single line of text. This chapter describes how to create and use entry-field controls in your PM applications.

About Entry Fields

An entry field provides the text-editing capabilities of a simple text editor and is useful whenever an application requires a short line of text from the user as illustrated in Figure 12-1.

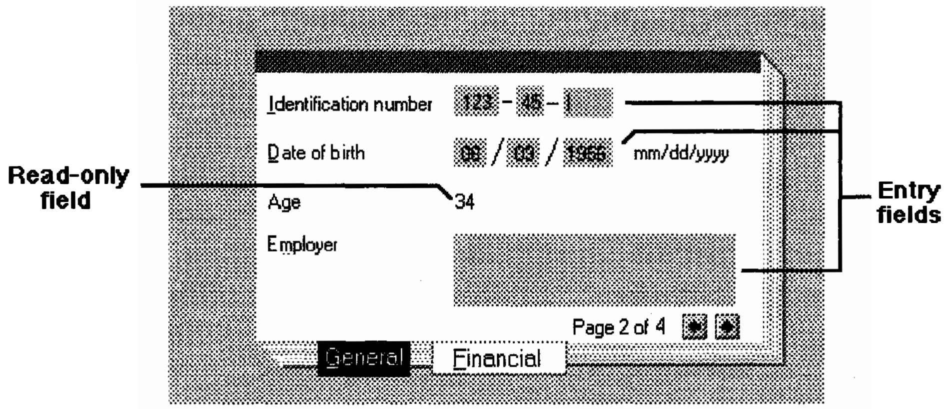


Figure 12-1. Example of Entry Fields

If the application requires more sophisticated text-editing capabilities and multiple lines of text from the user, the application can use a multiple-line entry (MLE) field. See Chapter 13, "Multiple-Line Entry Field Controls" on page 13-1 for more information about MLE controls.

Both the user and the application can edit text in an entry field. Applications typically use entry fields in dialog windows, although they can be used in non-dialog windows as well.

An application creates an entry field by specifying either the WC_ENTRYFIELD window class in the WinCreateWindow function or the ENTRYFIELD statement in a resource-definition file.

Entry-Field Styles

An entry field has a style that determines how it appears and behaves. An application specifies the style in either the WinCreateWindow function or the ENTRYFIELD statement in a resource-definition file. An application can specify a combination of the following styles for an entry field:

Table 12-1 (Page 1 of 2). Entry-Field Styles

Style	Description
ES_ANY	Allows the entry-field text to contain a mixture of double-byte and single-byte characters.

Table 12-1 (Page 2 of 2). Entry-Field Styles	
Style	**Description**
ES_AUTOSCROLL	Automatically scrolls text horizontally to show the insertion point.
ES_AUTOSIZE	Automatically sets the size of the entry field, based on the width of the field's text string and the metrics of the current system font. This style can set the width, height, or both—whichever has a value of -1 in the WinCreateWindow function or resource-definition file. This style affects only the initial size of the entry field; it does not adjust the size as the font or text-string width changes.
ES_AUTOTAB	Automatically moves the cursor to the next control window when the user enters the maximum number of characters.
ES_CENTER	Centers text within the entry field.
ES_DBCS	Specifies that the entry-field text consist of double-byte characters only.
ES_LEFT	Left-aligns text within the entry field.
ES_MARGIN	Draws a border around the entry field. The border is 1/2-character wide and 1/4-character high. Without this style, the application draws no border around the entry field. The width of the entry-field rectangle is increased on all sides by the width of this margin. After an entry field with the ES_MARGIN style is created, the WinQueryWindowRect function returns a larger rectangle that includes this margin and whose origin, therefore, is different from the origin specified when the entry field was created. If an application does not adjust for this size difference when moving or sizing an entry field, the entry field becomes larger after each moving and sizing operation.
ES_MIXED	Allows the entry-field text to contain a mixture of single-byte and double-byte characters. Unlike the ES_ANY style, this style lets ASCII DBCS data be converted to EBCDIC DBCS data without causing an overflow condition.
ES_READONLY	Prevents the user from entering or editing text in the entry field.
ES_RIGHT	Right-aligns text within the entry field.
ES_SBCS	Specifies that the entry-field text must consist of single-byte characters only.
ES_UNREADABLE	Displays each character as an asterisk (*). This style is useful when obtaining a password from the user.

Entry-Field Notification Codes

An entry field is always owned by another window. A WM_CONTROL notification message is sent to the owner whenever an event occurs in the entry field. This message contains a notification code that specifies the exact nature of the event. An entry field can send the following notification codes to its owner:

Table 12-2 (Page 1 of 2). Notification of Entry-Field Events	
Notification Code	**Description**
EN_CHANGE	Indicates that the contents of an entry field have changed.
EN_INSERTMODETOGGLE	Indicates that the insert mode has been toggled.

Table 12-2 (Page 2 of 2). Notification of Entry-Field Events	
Notification Code	**Description**
EN_KILLFOCUS	Indicates that an entry field has lost the keyboard focus.
EN_MEMERROR	Indicates that an entry field cannot allocate enough memory to perform the requested operation, such as extending the text limit.
EN_OVERFLOW	Indicates that either the user or the application attempted to exceed the text limit.
EN_SCROLL	Indicates that the text in an entry field is about to scroll.
EN_SETFOCUS	Indicates that an entry field received the keyboard focus.

An application typically ignores notification messages from an entry field, thus allowing default text editing to occur. For more specialized uses, an application can use notification messages to filter input. For example, if an entry field is intended for numbers only, an application can use the EN_CHANGE notification code to check the contents of the entry field each time the user enters a non-numeric character.

As an alternative, an application can prevent inappropriate characters from reaching an entry field by using EN_SETFOCUS and EN_KILLFOCUS, in filter code, placed in the main message loop. Whenever the entry field has the keyboard focus, the filter code can intercept and filter WM_CHAR messages before the WinDispatchMsg function passes them to the entry field. An application also can respond to certain keystrokes, such as the Enter key, as long as the entry-field control has the keyboard focus.

Default Entry-Field Behavior

The following table lists and describes all the messages specifically handled by the predefined entry-field control-window class (WC_ENTRYFIELD).

Table 12-3 (Page 1 of 3). Messages Handled by WC_ENTRYFIELD Class	
Message	**Description**
EM_CLEAR	Deletes the current text selection from the control window.
EM_COPY	Copies the current text selection to the system clipboard, in CF_TEXT format.
EM_CUT	Copies the current text selection to the system clipboard, in CF_TEXT format, and deletes the selection from the control window.
EM_PASTE	Copies the current contents of the system clipboard that have CF_TEXT format, replacing the current text selection in the control window.
EM_QUERYCHANGED	Returns TRUE if the text has changed since the last EM_QUERYCHANGED message.
EM_QUERYFIRSTCHAR	Returns the offset to the first character visible at the left edge of the control window.
EM_QUERYREADONLY	Determines whether the entry field is in the read-only state.

Table 12-3 (Page 2 of 3). Messages Handled by WC_ENTRYFIELD Class	
Message	**Description**
EM_QUERYSEL	Returns a long word that contains the offsets for the first and last characters of the current selection in the control window.
EM_SETFIRSTCHAR	Scrolls the text so that the character at the specified offset is the first character visible at the left edge of the control window.
EM_SETINSERTMODE	Toggles the text-entry mode between insert and overstrike.
EM_SETREADONLY	Sets the entry field to the read-only state.
EM_SETSEL	Sets the current selection to the specified character offsets.
EM_SETTEXTLIMIT	Allocates memory from the control heap for the specified maximum number of characters, returning TRUE if it is successful and FALSE if it is not. Failure causes the entry field to send a WM_CONTROL message with the EN_MEMERROR notification code to the owner window.
WM_ADJUSTWINDOWPOS	Changes the size of the control rectangle if the control has the ES_MARGIN style.
WM_BUTTON1DBLCLK	Occurs when the user presses mouse button 1 twice.
WM_BUTTON1DOWN	Sets the mouse capture and keyboard focus to the entry field, and prepares to track the movement of the mouse during WM_MOUSEMOVE messages.
WM_BUTTON1UP	Releases the mouse.
WM_BUTTON2DOWN	Returns TRUE to prevent this message from being processed further.
WM_BUTTON3DOWN	Returns TRUE to prevent this message from being processed further.
WM_CHAR	Handles text entry and other keyboard input events.
WM_CREATE	Validates the requested style and sets the window text.
WM_DESTROY	Frees the memory used for the window text.
WM_ENABLE	Sent when an application changes the enabled state of a window.
WM_MOUSEMOVE	If the mouse button is down, the entry field tracks the text selection. If the mouse button is up, the entry field sets the mouse pointer to the default arrow shape.
WM_PAINT	Draws the entry field and text.
WM_QUERYDLGCODE	Returns the predefined DLGC_ENTRYFIELD constant.
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.

Table 12-3 (Page 3 of 3). Messages Handled by WC_ENTRYFIELD Class	
Message	**Description**
WM_SETFOCUS	If the entry field is gaining the focus, it creates a cursor and sends the owner window a WM_CONTROL message with the EN_SETFOCUS notification code. If the entry field is losing the focus, it destroys the current cursor and sends the owner window a WM_CONTROL message with the EN_KILLFOCUS notification code.
WM_SETSELECTION	Toggles the current selection status.
WM_SETWINDOWPARAMS	Sets the specified window parameters, redraws the entry field, and sends the owner window a WM_CONTROL message with the EN_CHANGE notification code.
WM_TIMER	Blinks the insertion point if the entry field has the focus. The entry field scrolls the text, if necessary, while extending the selection to text that becomes visible in the window.

Entry-Field Text Editing

The user can insert (type) text or numeric values in an entry field when that entry field has the keyboard focus. An application can insert text by using the WinSetWindowText function. An application can insert numeric values by using the WinSetDlgItemShort function. The text or numeric value is inserted into the entry field at the cursor position.

The entry field's entry mode, either insert or overstrike, determines what happens when the user enters text. The user sets the entry mode by pressing the Insert key; the entry mode toggles each time the Insert key is pressed. The application can set the entry mode by sending the EM_SETINSERTMODE message to the entry field.

The cursor position, identified by a blinking bar, is specified by a character offset relative to the beginning of the text. The user can set the cursor position by using the mouse or the Arrow keys. An application can set the cursor position by using the EM_SETSEL message. This message directs the entry field to move the blinking bar to the given character position.

The EM_SETSEL message also sets the selection. The selection is one or more characters of text on which the entry field carries out an operation, such as deleting or copying to the clipboard. The user selects text by pressing the Shift key while moving the cursor, or by pressing mouse button 1 while moving the mouse. An application selects text by using the EM_SETSEL message to specify the cursor position and the anchor point. The selection includes all text between the cursor position and the anchor point. If the cursor position and anchor point are equal, there is no selection. An application can retrieve the selection (cursor position and anchor point) by using the EM_QUERYSEL message.

The user can delete characters, one at a time, by pressing the Delete key or the Backspace key. The Delete key deletes the character to the right of the cursor; the Backspace key deletes the character to the left of the cursor. The user also can delete a group of characters by selecting them and pressing the Delete key. An application can delete selected text by using the EM_CLEAR message.

An application can use the EM_QUERYCHANGED message to determine whether the contents of an entry field have changed.

An application can prevent the user from editing an entry field by setting the ES_READONLY style in the WinCreateWindow function or in the ENTRYFIELD statement in the resource-definition file. The application also can set and query the read-only state by using the EM_SETREADONLY and ES_QUERYREADONLY messages.

If text extends beyond the left or right edges of an entry field, the user can scroll the text by using the Arrow keys. An application can scroll the text by using the EM_SETFIRSTCHAR message to specify the first character visible at the left edge of the entry field. For scrolling to occur, the entry field must have the ES_AUTOSCROLL style. An application can use the EM_QUERYFIRSTCHAR message to obtain the first character that is currently visible.

Entry-Field Control Copy and Paste Operations

The user can cut, copy, and paste text in an entry field by using the Shift+Delete and Ctrl+Insert key combinations. An application, either by itself or in response to the user, can cut, copy, and paste text by using the EM_CUT, EM_COPY, and EM_PASTE messages. An application can use the ES_CUT and EM_COPY messages to copy the selected text to the clipboard. The EM_CUT message also deletes the text (EM_COPY does not). The EM_PASTE message copies the text on the clipboard to the current position in the entry field, replacing any existing text with the copied text. An application can delete the selected text, without copying it to the clipboard, by using the EM_CLEAR message.

Entry-Field Text Retrieval

An application can retrieve selected text from an entry field by calling WinQueryWindowText and then sending an EM_QUERYSEL message to retrieve the offsets to the first and last characters of the text selection. These offsets are used to retrieve selected text.

An application can retrieve numeric values by calling WinQueryDlgItemShort, passing the entry-field identifier and the handle of the owner window. WinQueryDlgItemShort converts the entry-field text to a signed or unsigned integer and returns the value in a specified variable. The application can use the WinWindowFromID function to retrieve the handle of the control window. The entry-field identifier is specified in the dialog template in the application's resource-definition file.

Using Entry-Field Controls

This section explains how to perform the following tasks:

- Create an entry field in a dialog or client window.
- Change the default size of the entry field.

Creating an Entry Field in a Dialog Window

A dialog window usually serves as the parent and owner of an entry field. The dialog window often includes a button that indicates whether the user wants to carry out an operation. When the user selects the button, the application queries the contents of the entry field and proceeds with the operation.

The definition of an entry field in an application's resource-definition file sets the initial text, window identifier, size, position, and style of the entry field. The following example shows how to define an entry field as part of a dialog template:

```
DLGTEMPLATE IDD_SAMPLE
BEGIN
    DIALOG "Sample Dialog", ID_DLG, 7, 7, 253, 145, FS_DLGBORDER,0
    BEGIN
        DEFPUSHBUTTON "~OK", DID_OK, 8, 151, 50, 23, WS_GROUP
        ENTRYFIELD "Here is some text", ID_ENTFLD, 42, 46, 68, 15,
            ES_MARGIN | ES_AUTOSCROLL
    END
END
```

Creating an Entry Field in a Client Window

To create an entry field in a non-dialog window, an application calls WinCreateWindow with the window class WC_ENTRYFIELD. The entry field is owned by an application's client window, whose window procedure receives notification messages from the entry field.

The following code fragment shows how to create an entry field in a client window:

```
#define ID_ENTRYFIELD 5

HWND hwnd, hwndEntryField1, hwndClient;
LONG xPos   =  50, yPos    = 100;
LONG xWidth = 100, yHeight =  20;

hwndEntryField1 = WinCreateWindow(
    hwndClient,             /* Parent-window handle */
    WC_ENTRYFIELD,          /* Window class         */
    "initial text",         /* Initial text         */
    WS_VISIBLE    |         /* Visible when created */
    ES_AUTOSCROLL |         /* Scroll text          */
    ES_MARGIN,              /* Create a border      */
    xPos, yPos,             /* x and y position     */
    xWidth, yHeight,        /* Width and height     */
    hwnd,                   /* Owner-window handle  */
    HWND_TOP,               /* Z-order position     */
    ID_ENTRYFIELD,          /* Window identifier    */
    NULL,                   /* No control data      */
    NULL);                  /* No pres. parameters  */
```

Figure 12-2. Code for Creating an Entry Field in a Client Window

Changing the Default Size of an Entry Field

The default text limit of an entry field is 32 characters. An application can set a non-default size when creating an entry field by setting the **cchEditLimit** member of an ENTRYFDATA structure and supplying a pointer to the structure as the *pCtlData* parameter to WinCreateWindow. The following code fragment creates an entry field with a text limit of 12 characters:

```
HWND hwndEntryField2;
HWND hwndClient;
ENTRYFDATA efd;
LONG xPos   = 50, yPos    = 50;
LONG xWidth = -1, yHeight = -1; /* must be -1 for ES_AUTOSIZE */

/* Initialize the ENTRYFDATA structure. */
efd.cb = sizeof(ENTRYFDATA);
efd.cchEditLimit = 12;
efd.ichMinSel = 0;
efd.ichMaxSel = 0;

/* Create the entry field. */
hwndEntryField2 = WinCreateWindow(
    hwndClient,             /* Parent-window handle */
    WC_ENTRYFIELD,          /* Window class         */
    "projects.xls",         /* No initial text      */
    WS_VISIBLE |            /* Visible when created */
    ES_MARGIN  |            /* Create a border.     */
    ES_AUTOSIZE,            /* System sets the size */
    xPos, yPos,             /* x and y positions    */
    xWidth, yHeight,        /* Width and height     */
    hwndClient,             /* Owner-window handle  */
    HWND_TOP,               /* Z-order position     */
    0,                      /* Window identifier    */
    &efd,                   /* Control data         */
    NULL);                  /* No pres. parameters  */
```

Figure 12-3. Code for Creating Entry Field with 12-Character Text Limit

To expand or reduce the text limit after creating the entry field, an application can send an EM_SETTEXTLIMIT message specifying a new maximum text limit for the entry field. The following code fragment increases to 20 characters the text limit of the entry field created in the previous example:

```
WinSendMsg(hwndEntryField2, EM_SETTEXTLIMIT,
    (MPARAM)20, (MPARAM)0);
```

Figure 12-4. Code for Creating Entry Field with 20-Character Text Limit

Retrieving Text From an Entry Field

An application can use the WinQueryWindowTextLength and WinQueryWindowText functions to retrieve the text from an entry field. WinQueryWindowTextLength returns the length of the text; WinQueryWindowText copies the window text to a buffer.

Typically, an application needs to retrieve the text from an entry field only if the user changes the text. An entry field sends an EN_CHANGE notification code in the low word of the first message parameter of the WM_CONTROL message whenever the text changes. The following code fragment sets a flag when it receives the EN_CHANGE code, checks the flag during the WM_COMMAND message and, if it is set, retrieves the text of the entry field:

```
HWND hwnd;
ULONG msg;
MPARAM mp1;
CHAR chBuf[64];
HWND hwndEntryField;
LONG cbTextLen;
LONG cbTextRead;
static BOOL fFieldChanged = FALSE;

switch (msg) {
    case WM_CONTROL:
        switch (SHORT1FROMMP(mp1)) {
            case IDD_ENTRYFIELD:

                /* Check if the user changed the entry-field text. */
                if ((USHORT) SHORT2FROMMP(mp1) == EN_CHANGE)
                    fFieldChanged = TRUE;
                return 0;
        }

    case WM_COMMAND:
        switch (SHORT1FROMMP(mp1)) {
            case DID_OK:

                /* If the user changed the entry-field text,        */
                /* obtain the text and store it in a buffer.        */
                if (fFieldChanged) {
                    hwndEntryField = WinWindowFromID(hwnd,
                        IDD_ENTRYFIELD);
                    cbTextLen = WinQueryWindowTextLength(hwndEntryField);
                    cbTextRead = WinQueryWindowText(hwndEntryField,
                        sizeof(chBuf), chBuf);
                    .
                    . /* Do something with the text.               */
                    .
                }
                WinDismissDlg(hwnd, 1);
                return 0;
        }
}
```

Figure 12-5. Code for Flagging a Text Change in an Entry Field

Summary

Following are the OS/2 functions, structures, and messages used with entry-field controls.

Table 12-4. Entry-Field Functions

Function Name	Description
WinQueryDlgItemShort	Converts the text of a dialog item into an integer value.
WinQueryWindowText	Copies window text into a buffer.
WinQueryWindowTextLength	Returns the length of the window text, excluding any NULL termination character.
WinSetDlgItemShort	Converts an integer value into the text of a dialog item.
WinSetWindowText	Sets the window text for a specified window.

Table 12-5. Entry-Field Structure

Structure Name	Description
ENTRYFDATA	Entry-field data structure

Table 12-6. Messages Sent to an Entry Field

Message	Description
EM_CLEAR	Deletes the text that forms the current selection.
EM_COPY	Sends the current selection to the clipboard.
EM_CUT	Sends the text that forms the current selection to the clipboard, then deletes it from the entry field control.
EM_PASTE	Replaces the text that forms the current selection with text from the clipboard.
EM_QUERYCHANGED	Queries whether the text of the entry field control has been changed since the last inquiry.
EM_QUERYFIRSTCHAR	Returns the zero-based offset of the first character displayed in the entry field control.
EM_QUERYREADONLY	Returns the read-only state of an entry field control.
EM_QUERYSEL	Gets the zero-based offsets of the bounds of the text that forms the current selection.
EM_SETFIRSTCHAR	Specifies the offset of the character to be displayed in the first position of the entry field control.
EM_SETINSERTMODE	Sets the insert mode of an entry field.
EM_SETREADONLY	Sets the read-only state of an entry field control.
EM_SETSEL	Sets the zero-based offsets of the bounds of the text that forms the current selection.
EM_SETTEXTLIMIT	Sets the maximum number of bytes that an entry field control can contain.

Table 12-7. Message Generated by an Entry Field to its Owner Window

Message	**Description**
WM_CHAR	Occurs when the user presses a key.
WM_CONTROL	Occurs when a control has a significant event to notify to its owner.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the entry field control window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the entry field control window parameters.

Chapter 13. Multiple-Line Entry Field Controls

A *multiple-line entry (MLE) field* is a sophisticated control window that enables a user to view and edit multiple lines of text. This chapter describes how to create and use multiple-line entry field controls in a PM application.

About Multiple-Line Entry Field Controls

An MLE field control gives an application the text-editing capabilities of a simple text editor. The application can create a multiple-line entry field by using the WinCreateWindow function or by specifying the MLE statement in a dialog-window template in a resource-definition file.

MLE Styles

The style of an MLE field control determines how the MLE field appears and behaves. An application can specify a combination of the following styles for an MLE field:

Table 13-1. Multiple-Line Entry Field Styles

Style	Description
MLS_BORDER	Draws a border around the MLE field.
MLS_HSCROLL	Adds a horizontal scroll bar to the MLE field. The MLE control enables this scroll bar whenever any line exceeds the width of the MLE field.
MLS_IGNORETAB	Directs the MLE control to ignore the Tab key.
MLS_READONLY	Prevents the MLE field from accepting text from the user. This style is useful for displaying lengthy static text in a client or dialog window.
MLS_VSCROLL	Adds a vertical scroll bar to the MLE field. The MLE control enables this scroll bar whenever the number of lines exceeds the height of the MLE field.
MLS_WORDWRAP	Automatically breaks lines that are longer than the width of the MLE field.

MLE Control Notification Codes

An MLE field control sends WM_CONTROL messages containing notification codes to its owner whenever certain events occur—for example, when the user or application tries to insert too much text, or when the user uses the scroll bars. The owner window uses the notification codes either to carry out custom operations for the MLE field or to respond to errors.

An MLE field control can send the following notification codes to its owner:

Table 13-2. Multiple-Line Entry Field Control Notification Codes

Code	Description
MLN_CHANGE	Indicates that the contents of the MLE field have changed.
MLN_CLPBDFAIL	Indicates that a clipboard operation failed.
MLN_HSCROLL	Indicates that the MLE text is about to scroll horizontally.
MLN_KILLFOCUS	Indicates that the MLE field lost the input focus.
MLN_MARGIN	Indicates that the mouse moved across the MLE field margin.
MLN_MEMERROR	Indicates that the MLE field control cannot allocate enough memory to perform the requested operation.
MLN_OVERFLOW	Indicates that the specified MLE operation would overflow the field's text limit or the format rectangle.
MLN_PIXHORZOVERFLOW	Indicates that the user entered more text than could fit horizontally in the MLE field.
MLN_PIXVERTOVERFLOW	Indicates that the user entered more text than could fit vertically in the MLE field.
MLN_SEARCHPAUSE	Indicates that the MLE field control paused during a search operation initiated by an MLM_SEARCH message.
MLN_SETFOCUS	Indicates that the MLE field received the input focus.
MLN_TEXTOVERFLOW	Indicates that the user or application attempted to exceed the text limit of the MLE field.
MLN_UNDOOVERFLOW	Indicates that the MLE field control cannot undo a text change because the undo operation involves too much text.
MLN_VSCROLL	Indicates that the MLE text is about to scroll vertically.

The MLE field control sends the MLN_HSCROLL or MLN_VSCROLL notification codes when the user enables the scroll bars so that the application can monitor the visible contents of the MLE field. The application also can monitor the contents of an MLE field by using the MLM_QUERYFIRSTCHAR message, which specifies the offset of the character in the upper-left corner of the MLE field. This represents the first MLE character that is visible to the user. To provide an alternative way of scrolling the contents of an MLE field, an application can move the character at the specified offset to the upper-left corner of an MLE field using the MLM_SETFIRSTCHAR message.

The MLE field control sends an MLN_CHANGE notification code when the user changes the text in some way. This notification code is especially useful when the MLE field is in a dialog window, because the dialog procedure can use this code to determine whether it should process the contents of the MLE field. If an application does not process MLN_CHANGE notification codes, it can use the MLM_QUERYCHANGED message to determine whether the user has made changes to the MLE text. The MLM_SETCHANGED message makes the MLE field control

send an MLN_CHANGE notification code with every event that occurs in the MLE field, regardless of whether the user has changed anything. This code also can be used to hide a change made by a user.

MLE Text Editing

An MLE field contains one or more lines of text. Each line consists of one or more characters and ends with one or more characters that represent the end of the line. The end-of-line characters are determined by the format of the text.

The user can type text in an MLE field when the MLE field has the focus. The application can insert text at any time by using the MLM_INSERT message and specifying the text as a null-terminated string. The MLE field control inserts the text at the cursor position or replaces the selected text.

The MLE field control entry mode, insert or overstrike, determines what happens when the user inserts text. The user sets the entry mode by pressing the Insert key. The entry mode alternates each time the user presses Insert. When overstrike mode is enabled, at least one character is selected. This means that the MLM_INSERT message always replaces at least one character. If insert mode is enabled, the MLM_INSERT message replaces only those characters the user or application has selected. Otherwise, the MLE field makes room for the inserted characters by moving existing characters to the right, starting at the cursor position.

The cursor position, identified by a blinking bar, is specified as a character offset relative to the beginning of the text. The user can set the cursor position by using the mouse or Arrow keys to move the blinking bar. An application can set the cursor position by using the MLM_SETSEL message, which directs the MLE field control to move the blinking bar to a given character position. The MLM_SETSEL message also can set the selection.

The *selection* is one or more characters of text on which the MLE field control carries out an operation, such as deleting or copying. The user selects text by pressing the Shift key while moving the cursor or by pressing mouse button 1 while moving the mouse. The user also can select a word in a block of text by double-clicking on the word. An application selects text by using the MLM_SETSEL message to specify the cursor position and the anchor point. The selection is all the text between the cursor position and the anchor point. If the cursor position and anchor point are equal, there is no selection. An application can retrieve the cursor position, anchor point, or both, by using the MLM_QUERYSEL message.

The user can delete characters, one at a time, by pressing the Delete key or the Backspace key. Pressing the Delete key deletes the character to the right of the cursor; pressing the Backspace key deletes the character to the left of the cursor and changes the cursor position. An application can delete one or more characters by using the MLM_DELETE message, which directs the MLE field control to delete a specified number of characters, starting at the given position. This message does not change the cursor position. An application can delete selected text by using the MLM_CLEAR message.

An application can reverse the previous operation by using the MLM_UNDO message, which restores the MLE field to its previous state. This is a quick way to fix editing mistakes. However, not all operations can be undone.

The application determines whether the previous operation can be undone by using the MLM_QUERYUNDO message, which returns TRUE and indicates the type of operation that can be undone. Using the MLM_RESETUNDO message, an application can prevent a subsequent MLM_UNDO message from changing the state of an MLE field.

MLE Text Formatting

An application can retrieve the number of lines of text in an MLE field by using the MLM_QUERYLINECOUNT message and can retrieve the number of characters in the MLE field by using the MLM_QUERYTEXTLENGTH message. The amount of text and, subsequently, the number of lines to be entered in an MLE field depend on the text limit. An application sets the text limit by using the MLM_SETTEXTLIMIT message and determines the current limit by using the MLM_QUERYTEXTLIMIT message. The user cannot set the text limit. If the user types to the text limit, the MLE field control beeps and ignores any subsequent keystrokes. If the application attempts to add text beyond the limit, the MLE field control truncates the text.

An application can control the length of each line in an MLE field by enabling word wrapping. When word wrapping is enabled, the MLE field control automatically breaks any line that is longer than the MLE field is wide. An application can set word wrapping by using the MLM_SETWRAP message, and it can determine whether the MLE field control is wrapping text by using the MLM_QUERYWRAP message. Word wrapping is disabled by default unless the application specifies the MLS_WORDWRAP style when creating the MLE field control.

An application can set tab stops for an MLE control by using the MLM_SETTABSTOP message. Tab stops specify the maximum width of a tab character. When the user or an application inserts a tab character, the MLE field control expands the character so that it fills the space between the cursor position and the next tab stop. The MLM_SETTABSTOP message sets the distance (in pels) between tab stops, and the MLE field control provides as many tab stops as necessary, no matter how long the line gets. An application can retrieve the distance between tab stops using the MLM_QUERYTABSTOP message.

An application can use the MLM_SETFORMATRECT message to set the *format rectangle* (MLE field). The format rectangle is used to set the horizontal and vertical limits for text. The MLE control sends a notification message to the parent window of the MLE field if text exceeds either of those limits. An application typically uses the format rectangle to provide its own word wrapping or other special text processing. An application can retrieve the current format rectangle by using the MLM_QUERYFORMATRECT message.

An application can prevent the user's editing of the MLE field by setting the MLS_READONLY style in the WinCreateWindow function or in the MLE statement in the resource-definition file. The application also can set and query the read-only state by using the MLM_SETREADONLY and MLM_QUERYREADONLY messages, respectively.

An application can set the colors and font for an MLE field by using the MLM_SETTEXTCOLOR, MLM_SETBACKCOLOR, and MLM_SETFONT messages. These messages affect all text in the MLE field. An MLE field cannot contain a mixture of fonts and colors. An application can retrieve the current values for the colors and font by using the MLM_QUERYTEXTCOLOR, MLM_QUERYBACKCOLOR, and MLM_QUERYFONT messages.

MLE Text Import and Export Operations

An application can copy text to and from an MLE field by importing and exporting. To import text to an MLE field, an application can use the MLM_IMPORT message, which copies text from a buffer to the MLE field. To export text from an MLE field, the application can use the MLM_EXPORT message, which copies text from the MLE field to a buffer. The application uses the MLM_SETIMPORTEXPORT message to set the import and export buffers.

An application can import and export text in a variety of formats. A text format, set with the MLM_FORMAT message, identifies which characters are used for the end-of-line characters. An MLE field can have the following text formats:

Table 13-3. Multiple-Line Entry Field Text Format

Format	Description
MLFIE_CFTEXT	Exported lines end with a carriage return/newline character pair (*0x0D, 0x0A*). Imported lines must end with a newline character, carriage return/newline character pair, or newline/carriage return character pair.
MLFIE_NOTRANS	Imported and exported lines end with a newline character (*0x0A*).
MLFIE_WINFMT	For exported lines, the carriage return/newline character pair marks a *hard* linebreak (a break entered by the user). Two carriage-return characters and a newline character (*0x0D, 0x0D, 0x0A*) mark a *soft* linebreak (a break inserted during word wrapping and not entered by the user). For imported lines, the extra carriage-return in soft linebreak characters is ignored.

The text format can affect the number of characters in a selection. To ensure that the export buffer is large enough to hold exported text, an application can send the MLM_QUERYFORMATLINELENGTH message. The application can send the MLM_QUERYFORMATTEXTLENGTH message to determine the number of bytes in the text to be exported.

Each time an application inserts text in an MLE field, the MLE field control automatically refreshes (repaints) the display by drawing the new text. When an application copies large amounts of text to an MLE field, refreshing can be quite time-consuming, so the application should disable the refresh state. The application disables the refresh state by sending the MLM_DISABLEREFRESH message. After copying all the text, the application can restore the refresh state by sending the MLM_ENABLEREFRESH message.

MLE Field Control Cut, Copy, and Paste Operations

The user can cut, copy, and paste text in an MLE field by using the Ctrl+Delete, Shift+Delete, and Shift+Insert key combinations. An application—either by itself or in response to the user—can cut, copy, and paste text by using the MLM_CUT, MLM_COPY, and MLM_PASTE messages. The MLM_CUT and MLM_COPY messages copy the selected text to the clipboard. The MLM_CUT message also deletes the text from the MLE field; MLM_COPY does not. The MLM_PASTE message copies the text from the clipboard to the current position in the MLE field, replacing any existing text with the copied text. An application can delete the selected text without copying it to the clipboard by using the MLM_CLEAR message.

An application also can copy the selected text from an MLE field to a buffer by using the MLM_QUERYSELTEXT message. This message does not affect the contents of the clipboard.

MLE Field Control Search and Replace Operations

An application can search for a specified string within MLE field text by using the MLM_SEARCH message, which searches for the string. The MLE field control returns TRUE if the string is found. The cursor does not move to the string unless the message specifies the MLFSEARCH_SELECTMATCH option.

An application also can use the MLM_SEARCH message to replace one string with another. If the message specifies the MLFSEARCH_CHANGEALL option, the MLE field control replaces all occurrences of the search string with the replacement string. Both the search string and the replacement string must be specified in an MLE_SEARCHDATA structure passed with the message.

Using Multiple-Line Entry Field Controls

This section explains how to create an MLE field control by using the WinCreateWindow function and by specifying the MLE statement in a dialog template in a resource-definition file.

Creating an MLE Field Control

The following code fragment shows how to create an MLE field control by using WinCreateWindow:

```
#define MLE_WINDOW_ID 2

HWND hwndParent;
HWND hwndMLE;

hwndMLE = WinCreateWindow(
    hwndParent,    /* Parent window     */
    WC_MLE,        /* Window class      */
    "Test",        /* Initial text      */
    WS_VISIBLE |   /* Window style      */
    MLS_BORDER,    /* Window style      */
    100, 100,      /* x and y positions */
    100, 100,      /* Width and height  */
    hwndParent,    /* Owner window      */
    HWND_TOP,      /* Top of z-order    */
    MLE_WINDOW_ID, /* Identifier        */
    NULL,          /* Control data      */
    NULL);         /* Pres. parameters  */
```

It also is common to create an MLE field control by using an MLE statement in a dialog-window template in a resource file, as shown in the following code fragment:

```
MLE "", IDD_MLETEXT, 110, 10, 50, 100,
    WS_VISIBLE | MLS_BORDER | MLS_WORDWRAP
```

The predefined class for an MLE control is WC_MLE. If you do not specify a style for the MLE control, the default styles used are MLS_BORDER, WS_GROUP, and WS_TABSTOP.

Importing and Exporting MLE Text

Importing and exporting MLE text takes place though a buffer. An *import* operation copies text from the buffer to the MLE field; an *export* operation copies text from the MLE to the buffer. Before an application can import or export MLE text, it must send an MLM_SETIMPORTEXPORT message to the MLE field control, specifying the address and size of the buffer.

To import text, an application sends the MLM_IMPORT message to the MLE field control. This message requires two parameters: *plOffset* and *cbCopy*. The *plOffset* parameter is a pointer to a variable that specifies the position in the MLE field where the text from the buffer is to be placed. The position is an *offset* from the beginning of the MLE text (that is, the number of characters from the beginning of the MLE text). If *plOffset* points to a variable that equals -1, the MLE field control places the text starting at the current cursor position. On return, this variable contains the offset to the first character beyond the imported text. The *cbCopy* parameter of the MLM_IMPORT message points to a variable that specifies the number of bytes to import. The following code fragment reads text from a file to a buffer, then imports the text to an MLE field:

```
HWND hwndMle;
CHAR szMleBuf[512];
IPT lOffset = 0;
PSZ pszTextFile;
HFILE hf;
ULONG cbCopied;
ULONG ulAction;
ULONG cbBytesRead;

.
. /* Obtain a file name from the user.                              */
.
  /* Open the file.                                                */

DosOpen(pszTextFile, &hf, &ulAction, 0, FILE_NORMAL,
    FILE_OPEN | FILE_CREATE, OPEN_ACCESS_READONLY |
    OPEN_SHARE_DENYNONE, NULL);

/* Zero-fill the buffer using memset, a C run-time function.      */
memset(szMleBuf, 0, sizeof(szMleBuf));

/* Set the MLE import-export buffer.                               */
WinSendMsg(hwndMle, MLM_SETIMPORTEXPORT, MPFROMP(szMleBuf),
    MPFROMSHORT ((USHORT) sizeof(szMleBuf)));

/*
 * Read the text from the file to the buffer, then import it
 * to the MLE.
 */

do {
    DosRead(hf, szMleBuf, sizeof(szMleBuf), &cbBytesRead);
    cbCopied = (ULONG) WinSendMsg(hwndMle, MLM_IMPORT,
        MPFROMP( &lOffset), MPFROMP(&cbBytesRead));
} while (cbCopied);

/* Close the file.                                                 */
DosClose(hf);
```

To export MLE text, an application sends the MLM_EXPORT message to the MLE control. Like MLM_IMPORT, the MLM_EXPORT message takes the *plOffset* and *cbCopy* parameters. The *plOffset* parameter is a pointer to a variable that specifies the offset to the first character to export. A value of -1 specifies the current cursor position. On return, the variable contains the offset to the first character in the MLE field not copied to the buffer. The *cbCopy* parameter is a pointer to a variable that specifies the number of bytes to export. On return, this variable equals 0 if the number of characters actually copied does not exceed the number specified to be copied. The following code fragment shows how to export text from an MLE field, then store the text in a file:

```
HWND hwndMle;
CHAR szMleBuf[512];
IPT lOffset = 0;
PSZ pszTextFile;
HFILE hf;
ULONG cbCopied;
ULONG ulAction;
ULONG cbBytesWritten;
ULONG cbCopy;

/* Zero-fill the buffer using memset, a C run-time function.  */
memset(szMleBuf, 0, sizeof(szMleBuf));

/* Set the MLE import-export buffer.                           */
WinSendMsg(hwndMle, MLM_SETIMPORTEXPORT, MPFROMP(szMleBuf),
    MPFROMSHORT ((USHORT) sizeof(szMleBuf)));
    .
    . /* Obtain a filename from the user.                      */
    .

/* Open the file.                                              */
DosOpen(pszTextFile, &hf, &ulAction, 0, FILE_NORMAL,
    FILE_OPEN | FILE_CREATE, OPEN_ACCESS_WRITEONLY |
    OPEN_SHARE_DENYNONE, NULL);

/* Find out how much text is in the MLE.                       */
cbCopy = (ULONG) WinSendMsg(hwndMle, MLM_QUERYFORMATTEXTLENGTH,
    MPFROMLONG(lOffset), MPFROMLONG((-1)));

/* Copy the MLE text to the buffer.                            */
cbCopied = (ULONG) WinSendMsg(hwndMle, MLM_EXPORT,
    MPFROMP(&lOffset), MPFROMP(&cbCopy));

/* Write the contents of the buffer to the file.               */
DosWrite(hf, szMleBuf, sizeof(szMleBuf),
    &cbBytesWritten);

/* Close the file.                                             */
DosClose(hf);
```

Searching MLE Text

An application uses the MLM_SEARCH message and the MLE_SEARCHDATA structure to search for strings in MLE text. The first parameter of the MLM_SEARCH message is an array of flags that specify the style of the search. The application can set the MLFSEARCH_CASESENSITIVE flag if a case-sensitive search is required. If the application sets the MLFSEARCH_SELECTMATCH flag, the MLE field control highlights a matching string and, if necessary, scrolls the string into view. An application can use the MLFSEARCH_CHANGEALL flag to replace every occurrence of the string with the string specified in the *pchReplace* member of the MLE_SEARCHDATA structure.

The second parameter of the MLM_SEARCH message is a pointer to an MLE_SEARCHDATA structure that contains information required to perform the search operation. This structure includes a pointer to the string and, if the MLFSEARCH_CHANGEALL flag is set in the MLM_SEARCH message, a pointer to the replacement string. The *iptStart* and *iptStop* members specify the starting and ending positions of the search. These positions are specified as offsets from the beginning of the MLE field. A value of -1 in the *iptStart* member causes the search to start at the current cursor position. A negative value in the *iptStop* member causes the search to end at the end of the MLE field. If a matching string is found, the MLE field control returns the length of the string in the *cchFound* member.

The following code fragment uses an entry field to obtain a search string from the user, then searches an MLE field for an occurrence of the string. The search begins at the current cursor position and ends at the end of the MLE text. When the MLFSEARCH_SELECTMATCH flag is specified, the MLE field control highlights a matching string and scrolls it into view.

```
#define IDD_SEARCHFIELD  101

HWND hwnd;
HWND hwndEntryFld;
HWND hwndMle;
MLE_SEARCHDATA mlesrch;
CHAR szSearchString[64];

/*
 * Obtain the handle of the entry field containing the
 * search string.
 */
hwndEntryFld = WinWindowFromID(hwnd, IDD_SEARCHFIELD);

/* Obtain the search string from the entry field.            */
WinQueryWindowText(hwndEntryFld, sizeof(szSearchString),
    szSearchString);

/* Fill the MLE_SEARCHDATA structure.                        */
mlesrch.cb = sizeof(mlesrch);     /* Structure size          */
mlesrch.pchFind = szSearchString; /* Search string           */
mlesrch.pchReplace = NULL;        /* No replacement string   */
mlesrch.cchFind = 0;              /* Not used                */
mlesrch.cchReplace = 0;           /* Not used                */
mlesrch.iptStart = -1;            /* Start at cursor position */
mlesrch.iptStop = -1;             /* Stop at end of file     */

/* Start the search operation.                               */
WinSendMsg(hwndMle, MLM_SEARCH, MPFROMLONG(MLFSEARCH_SELECTMATCH),
    MPFROMP(&mlesrch));
```

Summary

Following are the OS/2 structures and messages used with multiple-line entry field controls.

Table 13-4. Multiple-Line Entry Field Control Structures

Structure Name	Description
MLECTLDATA	Multiple-line entry field control data structure.
MLEMARGSTRUCT	Multiple-line entry field margin information
MLEOVERFLOW	Multiple-line entry field overflow error structure.
MLE_SEARCHDATA	Multiple-line entry field search structure.

Table 13-5 (Page 1 of 2). Messages Received by an MLE Field Control

Message	Description
MLM_CHARFROMLINE	Returns the first insertion point on a given line.
MLM_CLEAR	Clears the current selection.
MLM_COPY	Copies the current selection to the clipboard.
MLM_CUT	Copies the text that forms the current selection to the clipboard, then deletes the text from the MLE field control.
MLM_DELETE	Deletes text.
MLM_DISABLEREFRESH	Disables screen refresh.
MLM_ENABLEREFRESH	Enables screen refresh.
MLM_EXPORT	Exports text to a buffer.
MLM_FORMAT	Sets the format to be used for buffer importing and exporting.
MLM_IMPORT	Imports text from a buffer.
MLM_INSERT	Deletes the current selection and replaces it with a text string.
MLM_LINEFROMCHAR	Returns the line number corresponding to a given insertion point.
MLM_PASTE	Replaces the text that forms the current selection with text from the clipboard.
MLM_QUERYBACKCOLOR	Queries the background color.
MLM_QUERYCHANGED	Queries the changed flag.
MLM_QUERYFIRSTCHAR	Queries the first visible character.
MLM_QUERYFONT	Queries which font is in use.
MLM_QUERYFORMATLINELENGTH	Returns the number of bytes to end of line after formatting is applied.
MLM_QUERYFORMATRECT	Queries the format dimensions and mode.
MLM_QUERYFORMATTEXTLENGTH	Returns the length of a specified range of characters after the current formatting is applied.
MLM_QUERYIMPORTEXPORT	Queries the current transfer buffer.

Table 13-5 (Page 2 of 2). Messages Received by an MLE Field Control	
Message	**Description**
MLM_QUERYLINECOUNT	Queries the number of lines of text.
MLM_QUERYLINELENGTH	Returns the number of bytes between a given insertion point and the end of line.
MLM_QUERYREADONLY	Queries the read-only mode.
MLM_QUERYSEL	Returns the location of the selection.
MLM_QUERYSELTEXT	Copies the currently selected text into a buffer.
MLM_QUERYTABSTOP	Queries the pel interval at which tab stops are placed.
MLM_QUERYTEXTCOLOR	Queries the text color.
MLM_QUERYTEXTLENGTH	Returns the number of characters in the text.
MLM_QUERYTEXTLIMIT	Queries the maximum number of bytes that a multiple-line entry field control can contain.
MLM_QUERYUNDO	Queries the possible undo or redo operations.
MLM_QUERYWRAP	Queries the wrap flag.
MLM_RESETUNDO	Resets the undo state to indicate the no undo operations are possible.
MLM_SEARCH	Searches for a specified text string.
MLM_SETBACKCOLOR	Sets the background color.
MLM_SETCHANGED	Sets or clears the changed flag.
MLM_SETFIRSTCHAR	Sets the first visible character.
MLM_SETFONT	Sets a font.
MLM_SETFORMATRECT	Sets the format dimensions and mode.
MLM_SETIMPORTEXPORT	Sets the current transfer buffer.
MLM_SETREADONLY	Sets or clears read-only mode.
MLM_SETSEL	Sets a selection.
MLM_SETABSTOP	Sets the pel interval at which tab stops are placed.
MLM_SETTEXTCOLOR	Sets the text color.
MLM_SETTEXTLIMIT	Sets the maximum number of bytes that a multiple-line entry field control can contain.
MLM_SETWRAP	Sets the wrap flag.
MLM_UNDO	Performs any available undo operations.

Table 13-6. Messages Issued by an MLE Field Control to Its Owner Window

Message	Description
WM_BUTTON1DBLCLK	Occurs when the user presses pointer button 1 twice within a specified time.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_BUTTON1UP	Occurs when the user releases pointer button 1.
WM_CHAR	Sent when the user presses a key.
WM_CONTROL	Occurs when an MLE field control has a significant event to notify to its owner.
WM_ENABLE	Sets the state of the MLE field.
WM_MOUSEMOVE	Occurs when the pointing device pointer moves.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the entry field control window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the entry field control window parameters.

Chapter 14. Scroll-Bar Controls

Scroll bars are control windows that convert mouse and keyboard input into integers; they are used by an application to scroll the contents of a client window. This chapter describes how to create and use scroll bars in PM applications.

About Scroll Bars

A scroll bar has three main parts: the bar, its arrows, and a slider (see Figure 14-1).

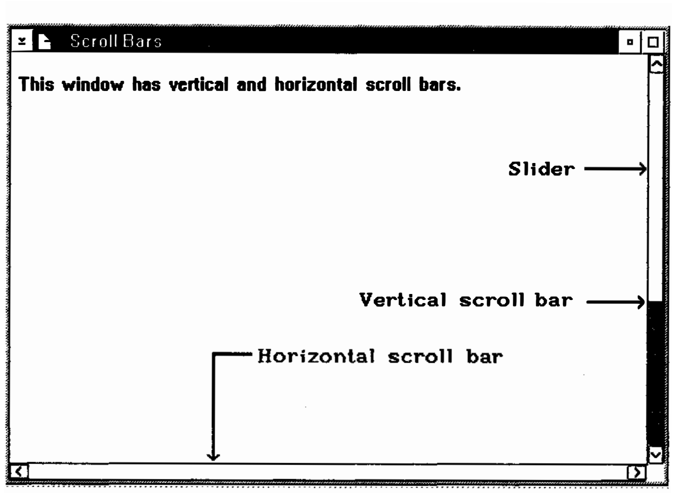


Figure 14-1. Scroll Bars in a Window

The arrows are located at each end of the scroll bar. The left scroll arrow, on the left side of a horizontal scroll bar, enables the user to scroll to the left in a document. The right scroll arrow lets the user scroll to the right.

On a vertical scroll bar, the upper scroll arrow enables the user to scroll upward in the document; the lower scroll arrow, downward. The slider, which lies between the two scroll arrows, reflects the current value of the scroll bar. Scroll bars monitor the slider and send notification messages to the owner window when the slider position changes as a result of mouse or keyboard input.

Although, typically, scroll bars are used in frame windows, an application can use stand-alone scroll bars of any size or shape, at any position, in a window of almost any class. Scroll bars can be used as parts of other control windows; for example, a list box uses a scroll bar to enable the user to view items when the list box is too small to display all the items.

Scroll-Bar Creation

An application can include a scroll bar in a standard frame window by specifying the FCF_HORZSCROLL or FCF_VERTSCROLL flag in the WinCreateStdWindow function. To create a scroll bar in another type of window, an application can specify the predefined (preregistered) window class WC_SCROLLBAR in the WinCreateWindow function or in the CONTROL statement in a resource file.

Although most applications specify an owner window when creating a scroll bar, an owner is not required. If an application does not specify an owner, the scroll bar does not send notification messages.

Scroll-Bar Styles

A scroll bar has styles that determine what it looks like and how it responds to input. Styles are specified in the WinCreateWindow function or the CONTROL statement. A scroll-bar can have the following styles:

Table 14-1. Scroll-Bar Styles

Style	Meaning
SBS_AUTOTRACK	Causes the entire slider to track the movement of the mouse pointer when the user scrolls the window. Without this style, only an outlined image of the slider tracks the movement of the mouse pointer, and the slider jumps to the new location when the user releases the mouse button.
SBS_HORZ	Creates a horizontal scroll bar.
SBS_THUMBSIZE	Causes the SBCDATA structure to store information used to calculate the size of the scroll-bar slider.
SBS_VERT	Creates a vertical scroll bar.

Scroll-Bar Range and Position

Every scroll bar has a range and a slider position. The range specifies the minimum and maximum values for the slider position. As the user moves the slider in a scroll bar, the scroll bar reports the slider position as an integer in this range. If the slider position is the minimum value, the slider is at the top of a vertical scroll bar or at the left end of a horizontal scroll bar. If the slider position is the maximum value, the slider is at the bottom or right end of the vertical or horizontal scroll bar, respectively.

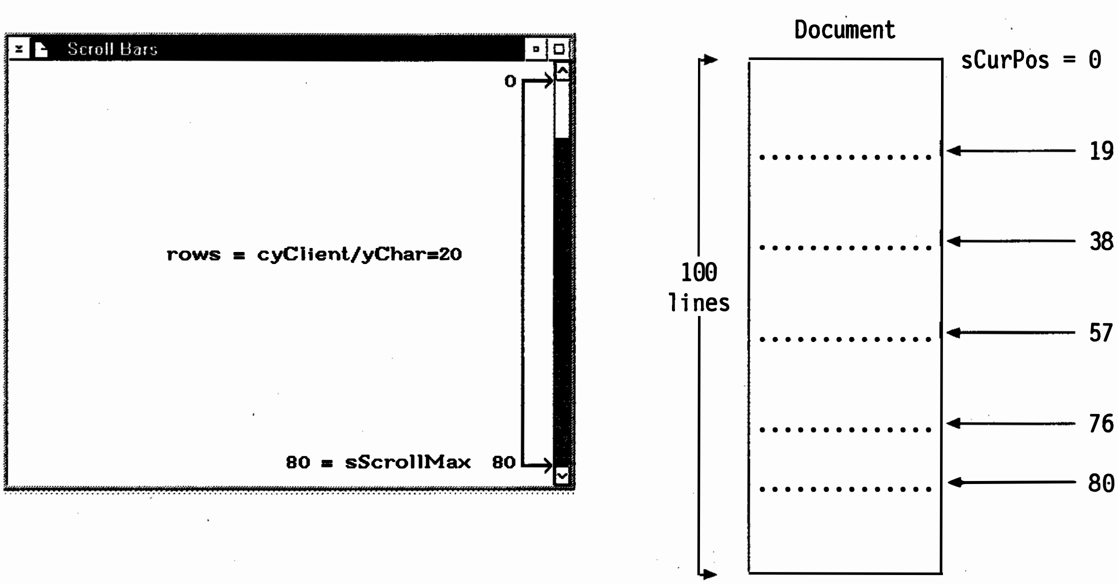


Figure 14-2. Determining Scroll-Bar Range

An application can adjust the range to convenient integers by using the SBM_SETSCROLLBAR message (or initially, by using the SBCDATA structure). This makes it easy to translate the slider position into a value that corresponds to the data being scrolled. For example, an application attempting to display 100 lines of text in a window that can show only 20 lines at a time could set the vertical scroll-bar range from 1 through 100. If the slider were at position 0, the first line

would be at the top of the window. If the slider were at position 100, the last line would be at the bottom of the window.

To establish a useful relationship between the scroll-bar range and the data, an application must adjust the range whenever the data or the size of the window changes. This means the application should adjust the range as part of processing WM_SIZE messages.

An application must move the slider in a scroll bar. Although the user requests scrolling in a scroll bar, the scroll bar does not update the slider position. Instead, it passes the request to the owner window, which scrolls the data and updates the slider position using the SBM_SETPOS message. The application controls the slider movement and can move the slider in the increments best suited for the data being scrolled.

An application can retrieve the current slider position of a scroll bar by sending the SBM_QUERYPOS message to the scroll bar.

If a scroll bar is a descendant of a frame window, its position relative to its parent can change when the position of the frame window changes. Frame windows draw scroll bars relative to the upper-left corner of the frame window (rather than the lower-left corner). The frame window can adjust the *y* coordinate of the scroll-bar position, which would be desirable if the scroll bar is a child of the frame window, but would be undesirable if the scroll bar is not a child window.

Scroll-Bar Notification Messages

A scroll bar sends notification messages to its window whenever the user clicks the scroll bar. WM_VSCROLL and WM_HSCROLL are the notification messages for vertical and horizontal scroll bars, respectively. If the scroll bar is a frame control window, the frame window passes the message to its client window.

Each notification message includes the scroll-bar identifier, scroll-bar command code corresponding to the action of the user, and, in some cases, the position of the slider. If an application creates a scroll bar as part of a frame control window, the scroll-bar identifier is the predefined constant FID_VERTSCROLL or FID_HORZSCROLL. Otherwise, it is the identifier given in the WinCreateWindow function.

The scroll-bar command codes specify the action the user has taken. Operating system user-interface guidelines recommend certain responses for each action. Figure 14-3 on page 14-4 illustrates the SBM_*xxx* messages your application can send to a scroll bar.

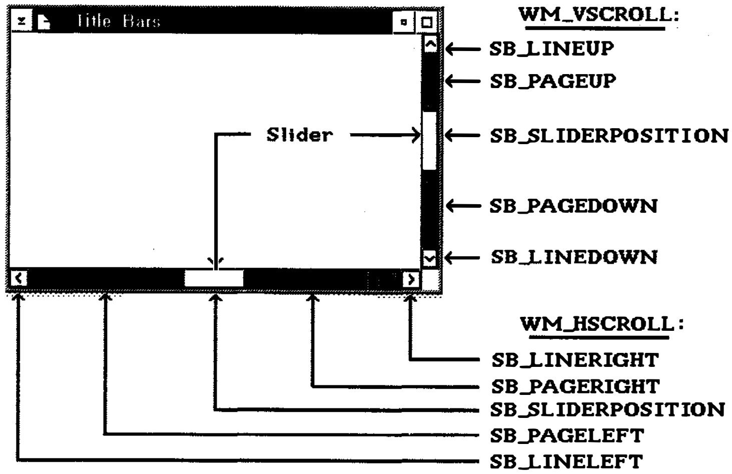


Figure 14-3. Standard Window Scroll Bar and Command Codes

Following is a list of the command codes; for each code, the user action is specified, followed by the application's response. In each case, a scrolling unit, appropriate for the given data, must be defined by the application. For example, for scrolling text vertically, the typical unit is a line.

Table 14-2 (Page 1 of 2). Scroll-Bar Command Codes

Command Code	Description
SB_LINEUP	Indicates that the user clicked the top scroll arrow. Decrement the slider position by one, and scroll toward the top of the data by one unit.
SB_LINEDOWN	Indicates that the user clicked the bottom scroll arrow. Increment the slider position by one, and scroll toward the bottom of the data by one unit.
SB_LINELEFT	Indicates that the user clicked the left scroll arrow. Decrement the slider position by one, and scroll toward the left end of the data by one unit.
SB_LINERIGHT	Indicates that the user clicked the right scroll arrow. Increment the slider position by one, and scroll toward the right end of the data by one unit.
SB_PAGEUP	Indicates that the user clicked the scroll-bar background above the slider. Decrement the slider position by the number of data units in the window, and scroll toward the top of the data by the same number of units.
SB_PAGEDOWN	Indicates that the user clicked the scroll-bar background below the slider. Increment the slider position by the number of data units in the window, and scroll toward the bottom of the data by the same number of units.
SB_PAGELEFT	Indicates that the user clicked the scroll-bar background to the left of the slider. Decrement the slider position by the number of data units in the window, and scroll toward the left end of the data by the same number of units.

Table 14-2 (Page 2 of 2). Scroll-Bar Command Codes	
Command Code	**Description**
SB_PAGERIGHT	Indicates that the user clicked the scroll-bar background to the right of the slider. Increment the slider position by the number of data units in the window, and scroll toward the right end of the data by the same number of units.
SB_SLIDERTRACK	Indicates that the user is dragging the slider. Applications that draw data quickly can set the slider to the position given in the message, and scroll the data by the same number of units the slider has moved. Applications that cannot draw data quickly should wait for the SB_SLIDERPOSITION code before moving the slider and scrolling the data.
SB_SLIDERPOSITION	Indicates that the user released the slider after dragging it. Set the slider to the position given in the message, and scroll the data by the same number of units the slider was moved.
SB_ENDSCROLL	Indicates that the user released the mouse after holding it on an arrow or in the scroll-bar background. No response is necessary.

If the command code is SB_SLIDERTRACK or SB_SLIDERPOSITION, indicating that the user is moving the scroll-bar slider, the notification message also contains the current position of the slider.

The owner window can send a message to the scroll bar to read or reset the current value and range of the scroll bar. To reflect any changes in the state of the scroll bar, the owner window also can adjust the data the scroll bar controls.

An application can use the WinEnableWindow function to disable a scroll bar. A disabled scroll bar ignores the actions of the user, sending out no notification messages when the user tries to manipulate it. If an application has no data to scroll, or if all data fits in the client window, the application should disable the scroll bar.

Scroll Bars and the Keyboard

When a scroll bar has the keyboard focus, it generates notification messages for the following keys:

Table 14-3. Scroll-bar Notification Messages	
Keys	**Response**
UP	SB_LINEUP or SB_LINELEFT
LEFT	SB_LINEUP or SB_LINELEFT
DOWN	SB_LINEDOWN or SB_LINERIGHT
RIGHT	SB_LINEDOWN or SB_LINERIGHT
PGUP	SB_PAGEUP or SB_PAGELEFT
PGDN	SB_PAGEDOWN or SB_PAGERIGHT

If an application uses scroll bars to scroll data but does not give the scroll bar the input focus, the window with the focus must process keyboard input. The window can generate scroll-bar notification messages or carry out the indicated scrolling. The following table shows the responses to keys that a window must process:

Table 14-4. Focus Window Message Responses to Keys

Key	**Response**
UP	SB_LINEUP
DOWN	SB_LINEDOWN
PGUP	SB_PAGEUP
PGDN	SB_PAGEDOWN
CTRL+HOME	SB_SLIDERTRACK, with the slider set to the minimum position
CTRL+END	SB_SLIDERTRACK, with the slider set to the maximum position
LEFT	SB_LINELEFT
RIGHT	SB_LINERIGHT
CTRL+PGUP	SB_PAGELEFT
CTRL+PGDN	SB_PAGERIGHT
HOME	SB_SLIDERTRACK, with the slider set to the minimum position
END	SB_SLIDERTRACK, with the slider set to the maximum position

For vertical scroll bars that are part of list boxes, the following table shows the responses to keys:

Table 14-5. List Box Responses to Keys

Key	**Command**
CTRL+UP	SB_SLIDERTRACK, with the slider set to the minimum position
CTRL+DOWN	SB_SLIDERTRACK, with the slider set to the maximum position
F7	SB_PAGEUP
F8	SB_PAGEDOWN

Using Scroll Bars

This section explains how to perform the following tasks:

- Create scroll bars.
- Retrieve a scroll-bar handle.
- Initialize, adjust, and read the scroll-bar range and position.

Creating Scroll Bars

When creating a frame window, you can add scroll bars by specifying the FCF_HORZSCROLL flag, FCF_VERTSCROLL flag, or both flags in the WinCreateStdWindow function. This adds horizontal, vertical, or both (as specified) scroll bars to the frame window. The frame window owns the scroll bars and passes notification messages from the scroll bars to the client window.

The following code fragment adds scroll bars to a frame window:

```
/* Set flags for a main window with scroll bars. */
ULONG ulFrameControlFlags =
    FCF_STANDARD | FCF_HORZSCROLL | FCF_VERTSCROLL;

/* Create the window.                              */
hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
    WS_VISIBLE,
    &ulFrameControlFlags,
    szClientClass,
    szFrameTitle,
    0,
    (HMODULE) NULL,
    0,
    &hwndClient);
```

Scroll bars created this way have the window identifier FID_HORZSCROLL or FID_VERTSCROLL. To determine the size and position of the scroll bars, the frame window uses the standard size specified by the system values SV_CXVSCROLL and SV_CYHSCROLL. The position always is defined by the right and bottom edges of the frame window.

Another way to create scroll bars is using the WinCreateWindow function. This method is most commonly used for stand-alone scroll bars. Creating scroll bars this way lets you set the size and position of the scroll bars. You also can specify which window should receive notification messages.

The following code fragment creates a stand-alone scroll bar:

```
#define ID_SCROLL_BAR 1

HWND hwndScroll,hwndClient;
hwndScroll = WinCreateWindow(
    hwndClient,                  /* Scroll-bar parent window      */
    WC_SCROLLBAR,                /* Preregistered scroll-bar class */
    (PSZ) NULL,                  /* No window title               */
    SBS_VERT | WS_VISIBLE,       /* Vertical style and visible    */
    10, 10,                      /* Position & Size               */
    20, 100,                     /* Size                          */
    hwndClient,                  /* Owner                         */
    HWND_TOP,                    /* Z-order position              */
    ID_SCROLL_BAR,               /* Scroll-bar identifier         */
    NULL,                        /* No class-specific data        */
    NULL);                       /* No presentation parameters    */
```

Retrieving a Scroll-Bar Handle

If you use the WinCreateStdWindow function to create a scroll bar as a child of the frame window, you must be able to retrieve the scroll-bar handle. One way to do this is to use the WinWindowFromID function, the frame-window handle, and a predefined identifier (such as FID_HORZSCROLL or FID_VERTSCROLL), as shown in the following code fragment:

```
HWND hwndFrame,hwndHorzScroll,hwndVertScroll;

hwndHorzScroll = WinWindowFromID(hwndFrame, FID_HORZSCROLL);
hwndVertScroll = WinWindowFromID(hwndFrame, FID_VERTSCROLL);
```

If the standard frame window includes a client window, you can use that handle to access the scroll bars. The idea is to get the frame-window handle first; then, the scroll-bar handle.

```
HWND hwndScroll,hwndClient;

/* Get a handle to the horizontal scroll bar. */
hwndScroll = WinWindowFromID(
    WinQueryWindow(hwndClient, QW_PARENT),
    FID_HORZSCROLL);
```

Using the Scroll-Bar Range and Position

You can initialize the current value and range of a scroll bar to non-default values by sending the SBCDATA structure with class-specific data for a call to WinCreateWindow:

```
#define ID_SCROLL_BAR 1

SBCDATA sbcd;
HWND hwndScroll,hwndClient;

/* Set up scroll-bar control data.                         */
sbcd.posFirst = 200;
sbcd.posLast  = 400;
sbcd.posThumb = 300;

/* Create the scroll bar.                                  */
hwndScroll = WinCreateWindow(hwndClient,
    WC_SCROLLBAR,
    (PSZ) NULL,
    SBS_VERT | WS_VISIBLE,
    10, 10,
    20, 100,
    hwndClient,
    HWND_TOP,
    ID_SCROLL_BAR,
    &sbcd,                     /* Class-specific data  */
    NULL);
```

You can adjust a scroll-bar value and range by sending it an SBM_SETSCROLLBAR message:

```
/* Set the scroll-bar value and range. */

WinSendMsg(hwndScroll, SBM_SETSCROLLBAR,
    (MPARAM)300,
    MPFROM2SHORT(200, 400));
```

You can read a scroll-bar value by sending it an SBM_QUERYPOS message:

```
USHORT usSliderPos;

/* Read the scroll-bar value. */
usSliderPos = (USHORT) WinSendMsg(hwndScroll,
    SBM_QUERYPOS, (MPARAM) NULL, (MPARAM) NULL);
```

Similarly, you can set a scroll-bar value by sending an SBM_SETPOS message:

```
/* Set the vertical scroll-bar value. */
WinSendMsg(hwndScroll, SBM_SETPOS, (MPARAM)300, (MPARAM) NULL);
```

You can read a scroll-bar range by sending it an SBM_QUERYRANGE message:

```
MRESULT mr;
USHORT  usMinimum, usMaximum;

/* Read the vertical scroll-bar range.                         */
mr = WinSendMsg(hwndScroll, SBM_QUERYRANGE, (MPARAM) NULL, (MPARAM) NULL);

usMinimum = SHORT1FROMMR(mr);            /* minimum in the low word  */
usMaximum = SHORT2FROMMR(mr);            /* maximum in the high word */
```

Summary

Following are the operating system structure and messages used with scroll bars.

Table 14-6. Scroll-Bar Structure

Structure name	Description
SBCDATA	Scroll-bar control data structure.

Table 14-7. Messages Sent to a Scroll Bar

Message	Description
SBM_QUERYPOS	Returns the slider position.
SBM_QUERYRANGE	Returns the scroll bar range.
SBM_SETPOS	Sets the position of the slider.
SBM_SETSCROLLBAR	Sets the scroll-bar range and slider positions.
SBM_SETTHUMBSIZE	Sets the scroll bar slider size.

Table 14-8. Messages Sent from a Scroll Bar to Its Owner Window

Message	Description
WM_HSCROLL	Occurs when a horizontal scroll bar control has a significant event to notify to its owner.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the scroll bar control window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the scroll bar control window.
WM_VSCROLL	Occurs when a vertical scroll bar control has a significant event to notify to its owner.

Chapter 15. Spin Button Controls

A *spin button* control (WC_SPINBUTTON window class) is a visual component that gives users quick access to a finite set of data by letting them select from a scrollable ring of choices. Since the user can see only one item at a time, a spin button should be used only with data that is intuitively related, such as a list of the months of the year, or an alphabetic list of cities or states. This chapter explains when and how to use spin buttons in PM applications.

About Spin Buttons

A *spin button* consists of at least one spin field that is a single-line entry (SLE) field, and up and down arrows that are stacked on top of one another. These arrows are positioned to the right of the SLE field. Figure 15-1 shows an example.

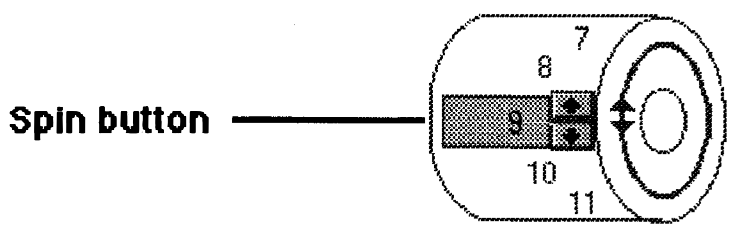


Figure 15-1. Example of a Spin Button

You can create multi-field spin buttons for those applications in which users must select more than one value. For example, in setting a date, the spin button control can provide individual fields for setting the month, day, and year. The first spin field in the spin button could contain a list of months; the second, a list of numbers; and the third, a list of years.

The application uses a multi-field spin button by creating one master component that contains a spin field and the spin arrows, and servant components that contain only spin fields. The spin buttons are created at component initialization. The servant components are passed a handle to the master component in a message. When a servant spin field has the focus, it is spun by the arrows in the master component.

The list of values in a spin button entry field can be an array of data or a list of consecutive integers, defined by an upper and a lower limit.

Creating a Spin Button

A spin button is created as a public window class by using the WinCreateWindow function, with a class style of WC_SPINBUTTON and a window style of WS_VISIBLE. These are joined with any of the spin button style flags by using a logical OR (|). The spin button style flags let you specify:

- Character input restrictions (*none, numeric, read-only*)
- Presentation of the data in the spin field (*left-justified, right-justified, centered*)
- Presence or absence of a border around the spin field

- Spin speed
- Zero-padding of numeric spin fields.

The placement and width of the spin button component are specified as parameters in the WinCreateWindow function.

The upper and lower limits of numeric fields, the value array pointer for arrays of strings, and the initial value in the spin field are all set by messages sent from the application to the component.

You can destroy the spin button component window using the WinDestroyWindow function when finished. The component handle that was returned when the spin button was created is the input parameter to the WinDestroyWindow function.

Figure 15-2 is an example of how to create a spin button.

```
ULONG      ulSpinStyle;                /* Spin Button style            */
HWND       hwndSpin;                   /* Spin Button window handle    */

/*********************************************************************/
/* Set the SPBS_* style flags.                                       */
/*********************************************************************/
ulSpinStyle    = SPBS_MASTER       |   /* Spinbtn has its own buttons, */
                 SPBS_NUMERICONLY  |   /* .. and it only holds numbers */
                 SPBS_JUSTRIGHT    |   /* .. that are right justified, */
                 SPBS_FASTSPIN;        /* .. and it spins faster as    */
                                       /*    the arrows are held down  */

/*********************************************************************/
/* Create the Spin Button control window.  The handle of the window  */
/* is returned in hwndSpin.                                          */
/*********************************************************************/
hwndSpin = WinCreateWindow (
                     hwndClient,       /* Parent window handle         */
                     WC_SPINBUTTON,    /* Spin Button window class name */
                     (PSZ)NULL,        /* No window text               */
                     ulSpinStyle,      /* Spin Button styles variable  */

                     (LONG)10,         /* X coordinate                 */
                     (LONG)10,         /* Y coordinate                 */
                     (LONG)150,        /* Window width                 */
                     (LONG)50,         /* Window height                */
                     hwndClient,       /* Owner window handle          */
                     HWND_TOP,         /* Sibling window handle        */
                     ID_SPINBUTTON,    /* Spin Button control window ID */
                     (PVOID)NULL       /* No control data structure    */
                     (PVOID)NULL);     /* No presentation parameters   */
```

Figure 15-2 (Part 1 of 2). Sample Code for Creating a Spin Button

```
/**********************************************************************/
/* Set the limits of the Spin Button control, since it has a style    */
/* of SPBS_NUMERICONLY.                                               */
/**********************************************************************/
WinSendMsg (hwndSpin,                 /* Spin Button window handle     */
            SPBM_SETLIMITS,           /* Set limits message            */
            (MPARAM)1000,             /* Spin Button maximum setting   */
            (MPARAM)0);               /* Spin Button minimum setting   */

/**********************************************************************/
/* Set the initial value of the Spin Button.                          */
/**********************************************************************/
WinSendMsg (hwndSpin,                 /* Spin Button window handle     */
            SPBM_SETCURRENTVALUE,     /* Set current value message.    */
            (MPARAM)100,              /* Spin Button initial value     */
            (MPARAM)NULL);            /* Reserved value                */

/**********************************************************************/
/* Because all items have been set, make the control visible.         */
/**********************************************************************/
WinShowWindow (hwndSpin,              /* Spin Button window handle     */
               TRUE);                 /* Make the window visible.      */
```

Figure 15-2 (Part 2 of 2). Sample Code for Creating a Spin Button

Graphical User Interface Support for Spin Buttons

Users can interact with the spin button using either the keyboard or a pointing device, such as a mouse, as follows:

- Using the select button (button 1) on the pointing device, users first give focus to the spin field they want to change, and then click on either the Up Arrow or Down Arrow until the value they want is displayed in the spin field.
- Using a keyboard, users press the:
  - Up Arrow and Down Arrow keys to see the choices
  - Left Arrow and Right Arrow keys to move the cursor left and right within a spin field
  - Home and End keys to move the cursor to the first and last characters in a spin field
  - Tab and BackTab (Shift+Tab) keys to move the input focus from one field to another in multi-field spin buttons.

Users can view the values in a spin field one at a time, or they can rapidly scroll a list by keeping either the Up or Down Arrow keys pressed. When a spin button is not read-only, users can advance quickly to the value they want to set in a spin field by typing over the value currently displayed.

Summary

Following are tables that describe the OS/2 spin button control notification codes, notification message, and window messages:

Table 15-1. Spin Button Control Notification Codes	
Code name	**Description**
SPBN_CHANGE	Sent when the contents of the spin field change.
SPBN_DOWNARROW	Sent when the Down Arrow button is clicked on or the Down Arrow key is pressed.
SPBN_ENDSPIN	Sent when the user releases the select button or one of the arrow keys while spinning a button.
SPBN_KILLFOCUS	Sent when the spin field loses the focus.
SPBN_SETFOCUS	Sent when the spin field is selected.
SPBN_UPARROW	Sent when the Up Arrow button is clicked on or the Up Arrow key is pressed.

Table 15-2. Spin Button Control Notification Message	
Message	**Description**
WM_CONTROL	Occurs when the spin button control has a significant event to notify to its owner.

Table 15-3. Spin Button Control Window Messages	
Message	**Description**
SPBM_OVERRIDESETLIMITS	Causes the component to set or reset numeric limits.
SPBM_QUERYLIMITS	Enables an application to query the limits of a numeric spin field.
SPBM_QUERYVALUE	Causes the component to show the value in the spin field.
SPBM_SETARRAY	Causes the component to set or reset the array of data.
SPBM_SETCURRENTVALUE	Causes the component to set or reset the current numeric value or array index.
SPBM_SETLIMITS	Causes the component to set or reset numeric limits.
SPBM_SETMASTER	Causes the component to identify its master.
SPBM_SETTEXTLIMIT	Sets the maximum number of characters allowed in a spin field.
SPBM_SPINDOWN	Causes the component to show the previous value (spin backward).
SPBM_SPINUP	Causes the component to show the next value (spin forward).

Chapter 16. Static Controls

A *static* control is a simple text field, bit map, or icon that an application can use to label, enclose, or separate other control windows. This chapter describes how to create and use static controls in a PM application.

About Static Controls

Unlike the other types of control windows, a static control does not accept user input nor send notification messages to its owner. The primary advantage of a static control is that it provides a label or graphic that requires little attention from an application. At most, an application might change the text or position of a static control.

Keyboard Focus

A static control never accepts the keyboard focus. When a static control receives a WM_SETFOCUS message, or when a user clicks the static control, the system advances the focus to the next sibling window that is not a static control. If the control has no siblings, the system gives the focus to the owner of the static control.

Static-Control Handle

Every static control is associated with a 32-bit data field. A static control with the SS_BITMAP or SS_ICON style uses this field to store the handle of the bit map or icon that it displays. An application can obtain that handle by sending the SM_QUERYHANDLE message to the control. An application can replace the bit map or icon by sending the SM_SETHANDLE message to the control, specifying a valid icon or bit map handle. Changing the handle causes the system to redraw the control.

For a non-icon or non-bit map static control, the data field is available for application-defined data and has no effect on the appearance of the control.

An application can retrieve the data field of a static-control window by calling WinWindowFromID, using the handle of the owner and the window identifier of the static control. The static-control window identifier is specified in either the dialog-window template or the WinCreateWindow function.

Static-Control Styles

A static control has style bits that determine whether the control displays text, draws a simple box containing text, displays an icon or a bit map, or shows a framed or unframed colored box. Applications can specify a combination of the following styles for a static control:

Table 16-1. Static-Control Styles

Style	Description
SS_BITMAP	Draws a bit map. The bit map resource must be provided in the resource-definition file. To include the bit map in a dialog window, the resource identifier must be specified in the *text* parameter of the CONTROL statement in the resource definition file. To include the bit map in a non-dialog window, the ASCII representation of the identifier must be specified in the *pszName* parameter of the WinCreateWindow function. That is, the first byte of the *pszName* parameter must be the cross-hatch character (#), and the remaining text must be the ASCII representation of the identifier (for example, **#125**).
SS_BKGNDFRAME	Creates a box whose frame has the background color.
SS_BKGNDRECT	Creates a rectangle filled with the background color.
SS_FGNDFRAME	Creates a box whose frame has the foreground color.
SS_FGNDRECT	Creates a rectangle filled with the foreground color.
SS_GROUPBOX	Creates a box whose upper-right corner contains control text. This style is useful for enclosing groups of radio buttons or check boxes in a box.
SS_HALFTONEFRAME	Creates a box whose frame has halftone shading.
SS_HALFTONERECT	Creates a box filled with halftone shading.
SS_ICON	Draws an icon. The resource identifier for the icon resource is determined the same way as the SS_BITMAP style. The icon resource must be in the resource-definition file.
SS_SYSICON	Draws a system-pointer icon. The resource identifier for the system-pointer resource is determined the same way as the SS_BITMAP style. To display this system pointer, the system calls WinQuerySysPointer with the specified identifier.
SS_TEXT	Creates a box with formatted text. An application can combine various formatting options with this style to produce formatted text in the boundaries of the control. The formatting flags are the same as those used for the WinDrawText function.

Default Static-Control Performance

The messages specifically handled by the predefined static-control class (WC_STATIC) are as follows:

Table 16-2 (Page 1 of 2). Messages Handled by WC_STATIC Class

Message	Description
SM_SETHANDLE	Sets the handle associated with the static control and invalidates the control window, forcing it to be redrawn.
SM_QUERYHANDLE	Returns the handle associated with the static-control window.
WM_ADJUSTWINDOWPOS	Adjusts the SWP structure so that the new window size matches the bit map, icon, or system-pointer dimensions associated with the static control.
WM_CREATE	Sets the text for a static-text control. Loads the bit map or icon resource for the bit map or icon static control. Returns TRUE if the resource cannot be loaded.
WM_DESTROY	Frees the text for a static-text control. Destroys the bit map or icon for a bit map or icon static control. The icon for a system-pointer static control is not destroyed because it belongs to the system.
WM_ENABLE	Invalidates the entire static-control window, forcing it to be redrawn.
WM_HITTEST	Returns the value HT_TRANSPARENT for the following static-control styles: SS_BKGNDFRAME SS_BKGNDRECT SS_FGNDFRAME SS_FGNDRECT SS_GROUPBOX SS_HALFTONEFRAME SS_HALFTONERECT. For other styles, this message returns the result of the WinDefWindowProc function.
WM_MATCHMNEMONIC	Returns TRUE if the mnemonic passed in the *mp1* parameter matches the mnemonic in the control-window text.
WM_MOUSEMOVE	Sets the mouse pointer to the arrow pointer and returns TRUE.
WM_PAINT	Draws the static control based on its style attributes.
WM_QUERYDLGCODE	Returns the predefined constant DLGC_STATIC.
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.
WM_SETFOCUS	Sets the focus to the next sibling window that can accept the focus; or if no such sibling exists, sets the focus to the parent window.

Table 16-2 (Page 2 of 2). Messages Handled by WC_STATIC Class	
Message	**Description**
WM_SETWINDOWPARAMS	Allows the text to be set (static-text controls only).

Using Static Controls

This section explains how to perform the following tasks:

- Include a static control in a dialog window.
- Include a static control in a client window.

Including a Static Control in a Dialog Window

To include a static control in a dialog window, you must define the control in a dialog-window template in a resource-definition file. The following resource-definition file creates a dialog window that contains a static-text control and three static-icon controls:

```
DLGTEMPLATE IDD_TOOLDLG LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "", IDD_TOOLDLG, 114, 53, 161, 127, FS_NOBYTEALIGN |
                FS_DLGBORDER | WS_VISIBLE | WS_SAVEBITS
    BEGIN
        CTEXT "Select a tool", IDS_TEXT, 49, 110, 56, 8,
                SS_TEXT | DT_CENTER | DT_TOP | WS_GROUP | WS_VISIBLE
        AUTORADIOBUTTON "Paintbrush", IDB_BRUSH, 63, 87, 61, 10,
                WS_TABSTOP | WS_GROUP | WS_VISIBLE
        AUTORADIOBUTTON "Scissors", IDB_SCISSORS, 63, 64, 60, 10,
                WS_TABSTOP | WS_VISIBLE
        AUTORADIOBUTTON "Eraser", IDB_ERASER, 65, 39, 43, 10,
                WS_TABSTOP | WS_VISIBLE
        ICON IDI_BRUSH, IDI_BRUSHICON, 33, 84, 22, 16,
                WS_GROUP | WS_VISIBLE
        ICON IDI_SCISSORS, IDI_SCISSORSICON, 33, 60, 22, 16,
                WS_GROUP | WS_VISIBLE
        ICON IDI_ERASER, IDI_ERASERICON, 33, 36, 22, 16,
                WS_GROUP | WS_VISIBLE
        PUSHBUTTON "OK", DID_OK, 10, 12, 38, 13, WS_TABSTOP |
                WS_GROUP | WS_VISIBLE
        PUSHBUTTON "Cancel", DID_CANCEL, 59, 12, 38, 13,
                BS_DEFAULT | WS_TABSTOP | WS_GROUP | WS_VISIBLE
        PUSHBUTTON "Help", IDB_HELP, 111, 13, 38, 13,
                BS_HELP | WS_TABSTOP | WS_GROUP | WS_VISIBLE
    END
END

ICON IDI_BRUSH brush.ico
ICON IDI_SCISSORS scissr.ico
ICON IDI_ERASER eraser.ico
```

Including a Static Control in a Client Window

An application can include a static control in a non-dialog window by calling WinCreateWindow with the window class WC_STATIC. The *flStyle* parameter to WinCreateWindow defines the appearance of the control.

The following code fragment creates a static text control whose size and position are based on the size of the client window and the metrics for the current font:

```
#define ID_TITLE 5

HWND hwnd,hwndStatic,hwndClient;
HPS hps;
RECTL rcl;
FONTMETRICS fm;
ULONG ulTitleLen;
CHAR szTitle[] = "Static Text Controls";

/* Obtain the size of the client window.              */
WinQueryWindowRect(hwnd, &rcl);

/* Obtain a presentation space handle and the metrics for
* the current font.                                   */
hps = WinBeginPaint (hwnd, (HPS) NULL, (PRECTL) NULL);
GpiQueryFontMetrics(hps, sizeof(FONTMETRICS), &fm);

/* Obtain the size of the static-control text string. */
ulTitleLen = (ULONG) strlen(szTitle);
/* Create the static control. Base the size and position
* on the size of the client window and the metrics of the
* current font.                                       */

hwndStatic = WinCreateWindow(
    hwndClient,                     /* Parent window          */
    WC_STATIC,                      /* Window class           */
    szTitle,                        /* Window text            */
    WS_VISIBLE |                    /* Make it visible        */
    SS_TEXT    |                    /* Static-text control    */
    DT_VCENTER |                    /* Center text vertically */
    DT_CENTER,                      /* Center text horizontally */
    ((rcl.xRight / 2) -
      (ulTitleLen / 2) * fm.lEmInc),/* x position             */
    rcl.yTop - fm.lEmHeight * 2,    /* y position             */
    fm.lEmInc * ulTitleLen,         /* Width                  */
    fm.lEmHeight * 2,               /* Height                 */
    hwndClient,                     /* Owner window           */
    HWND_TOP,                       /* Top of z-order         */
    ID_TITLE,                       /* Window identifier      */
    NULL,                           /* Control data           */
    NULL);                          /* Presentation parameters */

WinEndPaint(hps);
```

If your application creates a static control with the SS_ICON or SS_BITMAP style, make sure that the resource identifier specified in the *pszName* parameter corresponds to an icon or a bit map resource in the resource-definition file. If there is no resource, the application cannot create the static control.

Summary

Following are the operating system functions and messages used with static controls:

Table 16-3. Static-Control Functions

Function name	Description
WinQuerySysPointer	Returns the system pointer handle.
WinSetWindowPos	Allows the general positioning of a window.
WinSetWindowText	Sets the window text for a specified window.
WinWindowFromID	Returns the handle of the child window with the specified identity.

Table 16-4. Static-Control Messages

Message	Description
SM_QUERYHANDLE	Returns the icon or bit map handle of a static control.
SM_SETHANDLE	Sets the icon or bit map handle of a static control.
WM_MATCHMNEMONIC	Sent by the dialog box to a control window to determine whether a typed character matches a mnemonic in its window text.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the static control window procedure window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the static control window procedure window parameters.

Chapter 17. Title-Bar Controls

A *title-bar* is one of several control windows that comprise a standard frame window, giving the frame window its distinctive look and performance capabilities. This chapter describes how to create and use title-bar control windows in PM applications.

About Title Bars

The title bar in a standard frame window performs the following four functions:

- Displays the title of the window across the top of the frame window.
- Changes its highlighted appearance to show whether the frame window is active. (Ordinarily, the topmost window on the screen is the active window.)
- Responds to the actions of the user—for example, dragging the frame window to a new location on the screen.
- Flashes (as a result of the WinFlashWindow function) to get the attention of the user.

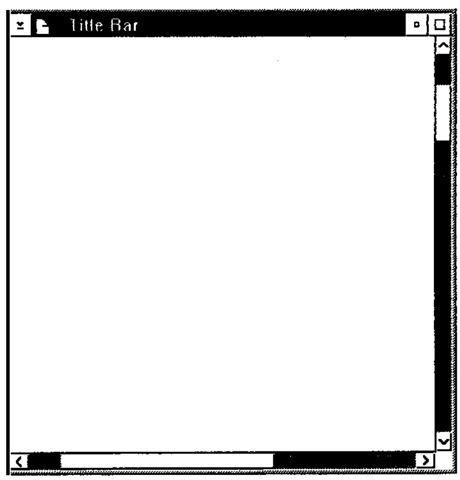


Figure 17-1. Title Bar in a Standard Frame Window

Once the frame controls are in place in the frame window, an application typically ignores them, because the system handles frame controls. In some cases, however, an application can take control of the title bar by sending messages to the title-bar control window.

Default Title-Bar Behavior

A title-bar control window sends messages to its owner (the frame window) when the control receives user input. Following are the messages that the title-bar control processes. Each message is described in terms of how the title-bar control responds to that message.

Table 17-1. Messages Processed by Title-Bar Control

Message	Description
TBM_QUERYHILITE	Returns the highlighted state of the title bar.
TBM_SETHILITE	Sets the highlighted state of the title bar, repainting the title bar if the state is changing.
WM_BUTTON1DBLCLK	Restores the title bar if the owner window is minimized or maximized. If the window is neither minimized nor maximized, this message maximizes the window.
WM_BUTTON1DOWN	Sends the WM_TRACKFRAME message to the owner window to start the tracking operation for the frame window.
WM_CREATE	Sets the text for the title bar. Returns FALSE if the text is already set.
WM_DESTROY	Frees the window text for the title bar.
WM_HITTEST	Always returns HT_NORMAL, so that the title bar does not beep when it is disabled. (It is disabled when the frame window is maximized.)
WM_PAINT	Draws the title bar.
WM_QUERYDLGCODE	Returns the predefined constant DLGC_STATIC. The user cannot use the Tab key to move to the title bar in a dialog window.
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.
WM_SETWINDOWPARAMS	Sets the specified window parameters.
WM_WINDOWPOSCHANGED	Returns FALSE. Processes this message to prevent the WinDefWindowProc function from sending the size and show messages.

Using Title-Bar Controls

This section explains how to:

- Include a title bar in a frame window.
- Alter the dragging action of a title bar.

Including a Title Bar in a Frame Window

An application can include a title bar in a standard frame window by specifying the FCF_TITLEBAR flag in the WinCreateStdWindow function.

The following code fragment shows how to create a standard frame window with a title bar, minimize and maximize (window-sizing) buttons, size border, system menu, and an application menu.

```
#define ID_MENU_RESOURCE 101

HWND hwndFrame,hwndClient;
UCHAR szClassName[255];

ULONG flControlStyle = FCF_TITLEBAR | FCF_MINMAX | FCF_SIZEBORDER |
                       FCF_SYSMENU  | FCF_MENU;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE | FS_ACCELTABLE,
                               &flControlStyle, szClassName, "",
                               0, (HMODULE) NULL, ID_MENU_RESOURCE,
                               &hwndClient);
```


```
#define ID_MENU_RESOURCE 101

HWND hwndFrame,hwndClient;
UCHAR szClassName[255];

ULONG flControlStyle = FCF_TITLEBAR | FCF_MINMAX | FCF_SIZEBORDER |
                       FCF_SYSMENU  | FCF_MENU;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE | FS_ACCELTABLE,
                               &flControlStyle, szClassName, "",
                               0, (HMODULE) NULL, ID_MENU_RESOURCE,
                               &hwndClient);
```


To get the window handle of a title-bar control, an application calls WinWindowFromID, specifying the frame-window handle and a constant identifying the title-bar control, as shown in the following code fragment:

```
hwndTitleBar = WinWindowFromID(hwndFrame, FID_TITLEBAR);
```

To set the text of a title bar, an application can use the WinSetWindowText function. The frame window passes the new text to the title-bar control in a WM_SETWINDOWPARAMS message.

Altering Dragging Action

When the user clicks the title bar, the title-bar control sends a WM_TRACKFRAME message to its owner (the frame window). When the frame window receives the WM_TRACKFRAME message, the frame sends a WM_QUERYTRACKINFO message to itself to fill in a TRACKINFO structure that defines the tracking parameters and boundaries. To modify the default behavior, an application must subclass the frame window, intercept the WM_QUERYTRACKINFO message, and modify the TRACKINFO structure. If the application returns TRUE for the WM_QUERYTRACKINFO message, the tracking operation proceeds according to the information in the TRACKINFO structure. If the application returns FALSE, no tracking occurs.

Summary

Following are the OS/2 functions, structures, and messages used with title-bar controls.

Table 17-2. Title-Bar Functions

Function name	**Description**
WinCreateStdWindow	Creates a standard window.
WinFlashWindow	Starts or stops the flashing of a window.
WinSetWindowText	Sets the window text for a specified window.
WinWindowFromID	Returns the handle of the child window with the specified identity.

Table 17-3. Title-Bar Structures

Structure name	**Description**
SWP	Set window position structure.
TRACKINFO	Tracking information structure.

Table 17-4. Title-Bar Messages

Message	Description
TBM_QUERYHILITE	Returns the highlighting state of a title-bar control.
TBM_SETHILITE	Used to highlight or unhighlight a title-bar control.
WM_BUTTON1DBLCLK	Occurs when the user presses button 1 of the pointing device twice.
WM_BUTTON1DOWN	Occurs when the user presses pointer button 1.
WM_CREATE	Occurs when an application requests the creation of a window.
WM_DESTROY	Occurs when an application requests the destruction of a window.
WM_HITTEST	Sent to determine which window is associated with an input from the pointing device.
WM_PAINT	Occurs when a window needs repainting.
WM_QUERYCONVERTPOS	Sent by an application to determine whether it is appropriate to begin conversion of DBCS characters.
WM_QUERYDLGCODE	Sent by the dialog manager to identify the type of control, to determine what kinds of messages the control understands, and to determine whether an input message can be processed by the dialog manager or passed down to the control.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the title-bar control window procedure window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the title-bar control window procedure window parameters.
WM_TRACKFRAME	Sent to a window whenever it is to be moved or sized.
WM_WINDOWPOSCHANGED	Sent to the window procedure of the window whose position is changed.

Chapter 18. Container Controls

A *container* control (WC_CONTAINER window class) is a visual component that holds objects. It provides a powerful and flexible component for easily developing products that conform to the Common User Access (CUA) user interface guidelines. This chapter describes the container control component and how to use it in PM applications.

About Container Controls

A container can display objects in various formats and views. Generally speaking, each view displays different information about each object. If a container's data is too large for the window's client area (hereinafter referred to as *work area* in accordance with CUA guidelines), scrolling mechanisms are enabled. The CUA direct manipulation protocol is fully supported, enabling a user to visually drag an object in a container window and drop it on another object or container window.

Containers are an integral component of the CUA user interface. For a complete description of CUA containers, refer to the *SAA CUA Guide to User Interface Design* and the *SAA CUA Advanced Interface Design Reference.*

Container Control Functions

The container control implements the following functions:

- Multiple types of views of a container's contents, such as:
  - Icon view
  - Name view
  - Text view
  - Tree view
  - Details view.
- Switching between container views quickly and easily
- Sharing records among multiple containers in the same process
- Displaying each view with a different font
- Directly editing container control text in all views, including blank text fields
- A split bar for vertically splitting the details view into two parts so that a user can widen one part to see more information
- Supporting various data types, such as:
  - Icons or bit maps for the icon, name, tree, and details views. In the details view, this includes the ability to use icons or bit maps in column headings as well as in the columns themselves.
  - Text that is supported in the following situations:
    - For container titles in all views
    - Beneath icons or bit maps in the icon view
    - To the right of icons or bit maps in the name and tree views
    - For any column or column heading in the details view
    - For container items in the text view.
  - Date, time, and number format, for container items in the details view.

- Direct manipulation
- Selection types, such as:
  - Single selection
  - Extended selection
  - Multiple selection.
- Selection techniques, such as:
  - Marquee selection
  - Two-swipe selections, such as:
    - Touch swipe
    - Range swipe.
  - First-letter selection.
- Selection mechanisms, such as:
  - Any pointing device
  - Keyboard.
- Multiple forms of emphasis:
  - In-use emphasis
  - Selected-state emphasis
  - Target emphasis.
- Ownerdraw, which enables an application to draw the container items instead of the container control's drawing them. In the details view, this can be done for each column.
- Sorting and filtering container items
- Arranging container items in the icon view, such as:
  - Automatic reposition mode that, when set, repositions container items as a result of inserting, removing, sorting, or filtering items, or changing window or font size
  - Arrange message mode that arranges overlapping icons or bit maps so that they no longer overlap.
- Scrolling a container's work area, such as:
  - When the current size of a container's work area is not large enough for all the container items to be visible
  - Dynamic scrolling to provide visible feedback, showing the movement of the container items relative to the position of the scroll box.
- Data caching:
  - To efficiently remove items from, and insert items in, a container as they scroll in and out of view.
- An option to optimize memory usage.

Container Control Basics

This section contains basic information about the container control that you need to understand before reading the remainder of the chapter. This important information is presented in the following order:

- Creating a container
- Understanding container items
- Allocating memory for container records and columns

- Understanding container views
- Changing a container view.

Creating a Container

You create a container by using the WC_CONTAINER window class name in the **ClassName** parameter of the WinCreateWindow function. Figure 18-1 shows the creation of the container. The styles specified in the ulCnrStyles variable (the CCS_* values) specifies that the container is to be created with the automatic positioning of container items and extended selection.

```
HWND hwndCnr;                      /* Container window handle          */
ULONG ulCnrStyles;                 /* Container window styles          */

/*******************************************************************/
/* Set CCS_* flags to customize the                                */
/* container.                                                      */
/*******************************************************************/
ulCnrStyles =
  CCS_AUTOPOSITION |                /* Auto position                    */
  CCS_ EXTENDSEL;                   /* Extended selection               */

/*******************************************************************/
/* Create the container control window.                            */
/*******************************************************************/
hwndCnr =
  WinCreateWindow(
    ClientHwnd,                    /* Parent window handle             */
    WC_CONTAINER,                  /* Container class name             */
    NULL,                          /* No window text                   */
    ulCnrStyles,                   /* Container styles                 */
    (LONG)10,                      /* X coordinate                     */
    (LONG)10,                      /* Y coordinate                     */
    (LONG)300,                     /* Window width                     */
    (LONG)200,                     /* Window height                    */
    ClientHwnd,                    /* Owner window handle              */
    HWND_TOP,                      /* Sibling window handle            */
    CONTAINER_ID,                  /* Container window ID              */
    NULL,                          /* No control data                  */
    NULL);                         /* No presentation parameters       */

/*******************************************************************/
/* Make the container control visible.                             */
/*******************************************************************/
WinShowWindow(
  hwndCnr,                         /* Container window handle          */
  TRUE);                           /* Make the window visible          */
```

Figure 18-1. Sample Code for Creating a Container

The container is created with a default set of control data, which can be changed using the CM_SETCNRINFO message. Refer to the *OS/2 2.0 Programming Reference* for a list of the default control data for the CNRINFO data structure.

Understanding Container Items

Container items can be anything that your application or a user might store in a container. Examples are executable programs, word processing files, graphics images, and database records.

Container item data is stored in RECORDCORE and MINIRECORDCORE data structures. Both the application and the container have access to the data stored in these records. Refer to the *OS/2 2.0 Programming Reference* for more information about the RECORDCORE and MINIRECORDCORE data structures.

The application is responsible for allocating memory for each record by using the CM_ALLOCRECORD message. See "Allocating Memory for Container Records" and "Allocating Memory for Container Columns" on page 18-5 for more information.

The maximum number of records is limited by the amount of memory in the user's computer. The container control does not limit the number of records that a container can have.

The following list shows which types of data can be displayed for each container view. See "Understanding Container Views" on page 18-5 for descriptions of the container views.

Table 18-1. Types of Container Views for Displaying Types of Data

View Types	Data
Icon	Icons or bit maps with text strings beneath.
Name	Icons or bit maps with text strings to the right.
Text	Text strings.
Tree	Icons or bit maps, and text strings.
Details	Icons or bit maps, text strings, numbers, times, and dates.

Allocating Memory for Container Records

Your application is required to allocate memory for a container record by using the CM_ALLOCRECORD message, which also enables you to allocate memory for additional application data. The sample code in Figure 18-2 shows how to allocate memory for one record. A pointer to the record is returned.

```
HWND        hwndCnr;            /* Container window handle          */
PRECORDCORE pRecord;            /* Pointer to RECORDCORE structure  */
ULONG       nRecords = 1;       /* 1 record to be allocated         */

pRecord =
  (PRECORDCORE)WinSendMsg(
    hwndCnr,                    /* Container window handle          */
    CM_ALLOCRECORD,             /* Message for allocating the record */
    (MPARAM)NULL,               /* No additional memory             */
    (MPARAM)nRecords);          /* Number of records to be allocated */
```

Figure 18-2. Sample Code for Allocating Memory for Container Records

Your application also can use the CM_ALLOCRECORD message to allocate memory for more than one container record. The application can request *n* container records with the *n***Records** parameter. If *n* is greater than one, the **pRecord**

parameter returns a pointer to the first record in a linked list of *n* records. Refer to the *OS/2 2.0 Programming Reference* for a description of the CM_ALLOCRECORD message and the RECORDCORE data structure.

Allocating Memory for Container Columns

In addition to allocating memory for records, an application also must allocate memory for columns of data if the details view is used. In the details view, a container's data is displayed in columns, each of which is described in a FIELDINFO data structure.

Memory is allocated for FIELDINFO data structures using the CM_ALLOCDETAILFIELDINFO message. Unlike the CM_ALLOCRECORD message, the CM_ALLOCDETAILFIELDINFO message does not allow the application to allocate memory for additional application data. However, the **pUserData** field of the FIELDINFO data structure can be used to store a pointer to the application-allocated data.

Multiple FIELDINFO data structures can be allocated with the **nFieldInfo** parameter of the CM_ALLOCDETAILFIELDINFO message. See "Details View" on page 18-14 for a description of the details view. Refer to the *OS/2 2.0 Programming Reference* for descriptions of the FIELDINFO data structure and the CM_ALLOCDETAILFIELDINFO message.

Understanding Container Views

When a user opens a container, the contents of that container are displayed in a window. A container window can present various views of its contents. Each view can provide different information about its container items. The container control provides the following views:

Table 18-2. Views of a Container's Contents

Type of View	Contents Displayed
Icon view	Displays either icons or bit maps, with text beneath the icons or bit maps, to represent container items. These are called icon/text or bit-map/text pairs. Each icon/text or bit-map/text pair represents one container item. This is the default view. See "Icon View" on page 18-6 for a description of the icon view.
Name view	Displays either icons or bit maps, with text to the right of the icons or bit maps, to represent container items. These are called icon/text or bit-map/text pairs. Each icon/text or bit-map/text pair represents one container item. See "Name View" on page 18-7 for a description of the name view.
Text view	Displays a simple text list to represent container items. See "Text View" on page 18-9 for a description of the text view.
Tree view	Displays a hierarchical view of the container items. Three types of tree views are available: tree text, tree icon, and tree name. See "Tree View" on page 18-10 for a description of the tree view.
Details view	Displays detailed information about each container item. The same type of data is displayed for each container item, arranged in columns. The data in each column can consist of an icon or bit map, text, numbers, dates, or times. See "Details View" on page 18-14 for a description of the details view.

The container control does not support both icons and bit maps in the same view. To specify whether icons or bit maps are used, an application can set either the CA_DRAWICON attribute or CA_DRAWBITMAP attribute, respectively, in the **flWindowAttr** field. The default is the CA_DRAWICON attribute. The size of the icon or bit map can be specified in the **slBitmapOrIcon** field. **flWindowAttr** and **slBitmapOrIcon** are fields of the CNRINFO data structure. Refer to the *OS/2 2.0 Programming Reference* for a description of the CNRINFO data structure.

If a text string is not specified for a view in a place where a text string could be used, a blank space is used as a placeholder. For example, if a text string is not placed beneath an icon in the icon view, a blank space is inserted just as though the text string was there. If this blank space is not a read-only field, the user can put text in the space by editing it directly. See "Direct Editing of Text in a Container" on page 18-31 for more information about editing text directly in a container control.

Icon View

The icon view (CV_ICON attribute) displays icon/text pairs or bit-map/text pairs to represent container items; this is the default. CV_ICON is an attribute of the CNRINFO data structure's **flWindowAttr** field.

In the icon view, icon/text pairs and bit-map/text pairs are icons and bit maps, respectively, with one or more lines of text displayed below each icon or bit map. Each line can contain one or more text characters, which are centered below the icon or bit map. The container control does not limit the number of lines or the number of characters in each line.

Generally, the icon or bit map contains an image that depicts the type of container item that it represents. For example, an icon or bit map that represents a bar chart might contain an image of a bar chart.

In the icon view, container items are positioned according to *x*- and *y*-coordinate positions. These are called *workspace coordinates*. You can supply these coordinates for each container item by using the **ptlIcon** field of the RECORDCORE data structure. See "Positioning Container Items" on page 18-28 for information about using workspace coordinates to position container items. Refer to the *OS/2 2.0 Programming Reference* for a description of the RECORDCORE data structure. Figure 18-3 provides an example of the icon view with various *x*- and *y*-coordinates specified in the **ptlIcon** field.

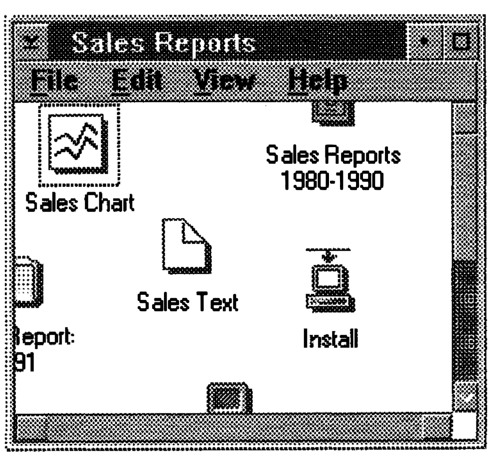


Figure 18-3. Icon View with Items Positioned at Workspace Coordinates

If you do not specify *x*- and *y*-coordinate positions, the container control positions the icons or bit maps at *(0,0)*. However, your application can arrange the icons or bit maps either by sending the CM_ARRANGE message or by setting the CCS_AUTOPOSITION style bit when creating a container. With both of these methods, the container items are arranged in rows, and any coordinates specified in the **ptlIcon** field are ignored.

The container items fill the topmost row until the width of the work area is reached. The container items then wrap to form another row immediately below the filled row. This process is repeated until all the container items are positioned in rows. Default spacing is implemented according to the guidelines for the CUA user interface. Figure 18-4 shows an example of the container after the CM_ARRANGE message was sent, or if the container was created with the CCS_AUTOPOSITION style bit set.

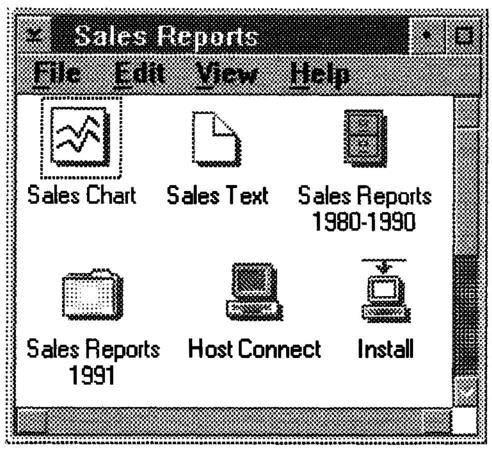


Figure 18-4. Icon View When Items Are Arranged or Automatically Positioned

If the CCS_AUTOPOSITION style bit is set and the container is displaying the icon view, container items are arranged automatically without the CM_ARRANGE message being sent when:

- The window size changes
- Container items are inserted, removed, sorted, invalidated, or filtered
- The font or font size changes.

In all of these cases, container items are arranged the same as when the CM_ARRANGE message is sent. The CCS_AUTOPOSITION style bit is valid only when it is used with the icon view.

If the CM_ARRANGE message is issued and the container control is not currently displaying the icon view, the container items are still arranged logically. Nothing changes in the current view; the arrangement of the container items is not visible until the user switches to the icon view.

Name View

The name view (CV_NAME attribute) displays icon/text or bit-map/text pairs to represent container items. CV_NAME is an attribute of the CNRINFO data structure's **flWindowAttr** field.

In the name view, icon/text pairs and bit-map/text pairs are icons and bit maps, respectively, with one or more lines of text displayed to the right of each icon or bit map. Each line can contain one or more text characters, which are left-justified.

The container control does not limit the number of lines or the number of characters in each line.

The container control offers the option of flowing or not flowing the container items in the name view. To *flow* container items means to dynamically arrange them in columns.

Non-Flowed Name View

If the container items are not flowed, the icon/text or bit-map/text pairs are placed in a single column in the leftmost portion of the work area, as in Figure 18-5.

Figure 18-5. Non-Flowed Name View

Flowed Name View

If the container items are flowed (CV_NAME | CV_FLOW), the container appears as in Figure 18-6. In this case, the container items fill the leftmost column until the depth of the work area is reached. The container items then wrap to form another column immediately to the right of the filled column. This process is repeated until all of the container items are positioned in columns.

The width of each column is determined by the widest text string within the column. The depth of the work area is determined by the size of the window.

Figure 18-6. Flowed Name View

Text View

The text view (CV_TEXT attribute) displays one or more lines of text to represent container items. CV_TEXT is an attribute of the CNRINFO data structure's **flWindowAttr** field.

Each line can contain one or more text characters, which are left-justified. The container control does not limit the number of lines or the number of characters in each line.

The container control offers the option of flowing or not flowing the container items in the text view.

Non-Flowed Text View

If the text strings are not flowed, the text for each container item is placed in a single column in the leftmost portion of the work area, as in Figure 18-7.

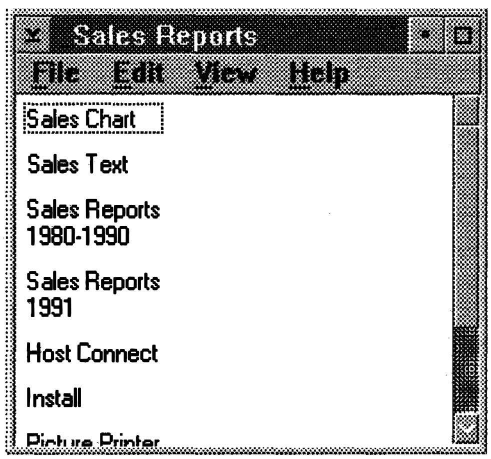


Figure 18-7. Non-Flowed Text View

Flowed Text View

If the text strings are flowed (CV_TEXT | CV_FLOW), the container appears as in Figure 18-8. In this case, the text strings fill the leftmost column until the depth of the work area is reached. The text strings then wrap to form another column immediately to the right of the filled column. This process is repeated until all the text strings are positioned in columns.

The width of each column is determined by the widest text string within the column. The depth of the work area is determined by the size of the window.

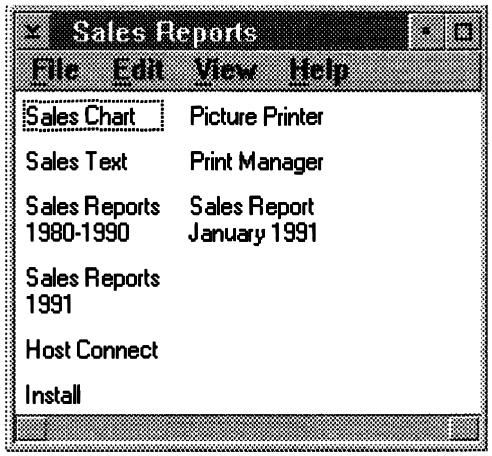


Figure 18-8. Flowed Text View

Tree View

The tree view (CV_TREE attribute) displays container items arranged hierarchically. CV_TREE is an attribute of the CNRINFO data structure's **flWindowAttr** field.

The leftmost items displayed in the tree view are at the *root level* and are the same items displayed in all the other container views. Items that contain other items are called *parent items*. The item or items that a parent item contains are called *child items* and can be displayed only in the tree view. Child items that contain other items serve a dual role: they are the children of their parent item, but they are parent items as well, with children of their own. For example, a parent item might be a book that contains individual child items for its chapters, or a folder that contains several reports. The chapters or reports, in turn, could be parent items that contain their own children, such as the major sections of a chapter or report.

If the child item or items of a parent item are not displayed, the parent item can be *expanded* to display them as a new branch in the tree view. Once a parent item has been expanded, it can be *collapsed* to remove its child items from the display.

You can use the **cxTreeIndent** and **cxTreeLine** fields of the CNRINFO data structure to specify the number of pels that a new branch is to be indented horizontally, and the width of the lines that are used to connect branches of the tree. These lines are displayed only if the CA_TREELINE attribute is specified in the **flWindowAttr** field.

The tree view has three different types: tree icon view, tree text view, and tree name view. Figure 18-9 uses the tree icon view to provide examples of root level, parent, and child items that were defined in this section. The expanded and collapsed bit maps shown in this figure are defined in the following section.

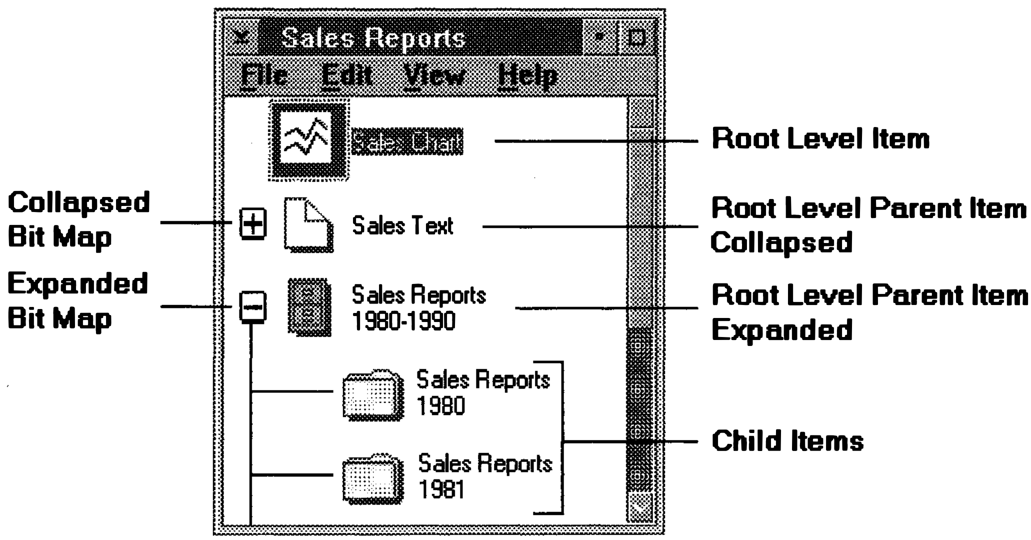


Figure 18-9. Sample Tree View Showing Root Level, Parent, and Child Items

Tree Icon View and Tree Text View

The tree icon and tree text views are identical in every aspect except one: their appearance on the screen. Container items in the tree icon view (CV_TREE | CV_ICON) are displayed as either icon/text pairs or bit-map/text pairs. The items are drawn as icons or bit maps with one or more lines of text displayed to the right of each icon or bit map. Figure 18-10 provides an example of a tree icon view that uses the default expanded and collapsed bit maps.

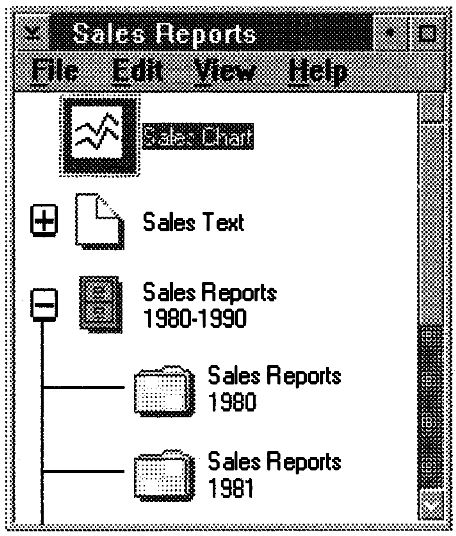


Figure 18-10. Tree Icon View

Container items in the tree text view (CV_TREE | CV_TEXT) are displayed as text strings. In both views, the container control does not limit the number of lines of text or the number of characters in each line. Figure 18-11 provides an example of the tree text view, again showing the default expanded and collapsed bit maps.

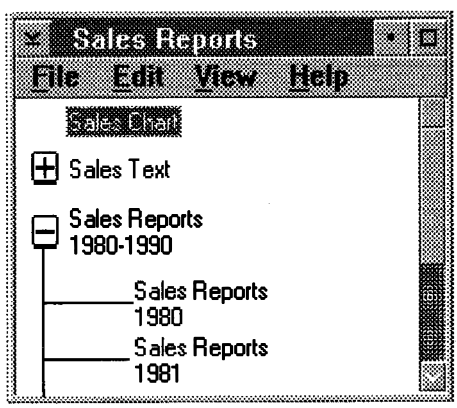


Figure 18-11. Tree Text View

In the tree icon and tree text views, a parent item is expanded by selecting the *collapsed* icon/bit map, which is displayed to the left of the parent item.

The collapsed icon/bit map should contain some visible indication that the item can be expanded. The default collapsed bit map that is provided by the container control uses a plus sign (+) to indicate that more items, the children of this parent, can be added to the view.

When the child items of a parent item are displayed, the collapsed icon/bit map to the left of that parent item changes to an expanded icon/bit map. Just as the collapsed icon/bit map provides a visible indication that an item can be expanded, so should the expanded icon/bit map indicate that an item can be collapsed. The default expanded bit map provided by the container control uses a minus sign (-) to indicate that the child items of this parent can be subtracted from the view. If any of the child items have children of their own, a collapsed or expanded icon/bit map is displayed to their immediate left as well.

To display your own collapsed and expanded icons or bit maps, specify their handles by using the **hptrCollapsed** and **hptrExpanded** fields of the CNRINFO data structure for icons, and the **hbmCollapsed** and **hbmExpanded** fields for bit maps. Also, you can use the **slTreeBitmapOrIcon** field to specify the size, in pels, of these collapsed and expanded icons and bit maps. Refer to the description of the CNRINFO data structure in the *OS/2 2.0 Programming Reference* for more information.

Tree Name View

Container items in the tree name view (CV_TREE | CV_NAME) are displayed as either icon/text pairs or bit-map/text pairs. Similar to the tree icon view, the items are drawn as icons or bit maps with one or more lines of text displayed to the right of each icon or bit map. The container control does not limit the number of lines or the number of characters in each line of text.

Unlike the tree icon view, however, separate collapsed and expanded icons/bit maps are not used. Instead, if an item is a parent, the icon or bit map that represents that item contains the same type of visible indication that is placed in a separate icon/bit map in the tree icon view to show that an item can be collapsed or expanded. In this way, the icon or bit map that represents the parent item can serve a dual purpose, and thus preserve space on the screen, an important consideration if the text strings used to describe items become too long.

The container control does not provide default icons or bit maps for the tree name view. To display your own collapsed and expanded icons or bit maps, specify their handles using the **hptrCollapsed** and **hptrExpanded** fields of the TREEITEMDESC data structure for icons, and the **hbmCollapsed** and **hbmExpanded** fields for bit maps. Also, you can use the **slBitmapOrIcon** field of the CNRINFO data structure to specify the size, in pels, of these collapsed and expanded icons and bit maps. Refer to the description of the TREEITEMDESC and CNRINFO data structures in the *OS/2 2.0 Programming Reference* for more information about these data structures and Figure 18-12 for an example of the tree name view.

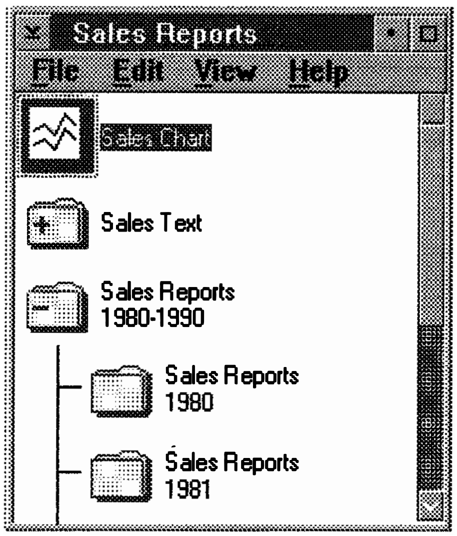


Figure 18-12. Tree Name View

Details View

The details view (CV_DETAIL attribute) of the container control can display the following data types to represent container items: icons or bit maps, text, numbers, dates, and times. CV_DETAIL is an attribute of the CNRINFO data structure's **flWindowAttr** field.

The data is arranged in columns, which can have headings. Each column can contain data that belongs to only one of the valid data types. Column headings can contain text, icons, or bit maps.

The width of each column can be explicitly specified in the **cxWidth** field of the FIELDINFO data structure. If a column width is not specified, it is determined by the widest entry in the column. Refer to the *OS/2 2.0 Programming Reference* for a description of the FIELDINFO data structure.

Columns can be inserted or removed dynamically. All of the columns in a given row represent a single container item; selecting the data portion of a row selects the entire row, not just the individual column.

Details view column headings and data can be top- or bottom-justified or vertically centered, as well as left- or right-justified or horizontally centered. In addition, horizontal separator lines can be specified between the column headings and the data; vertical separator lines can be placed between columns. In the example in Figure 18-13, Container Items, the icon, Description, and Item Size are the column headings.

Ownerdraw is supported for each column. See "Drawing Container Items and Painting Backgrounds" on page 18-34 for more information about ownerdraw.

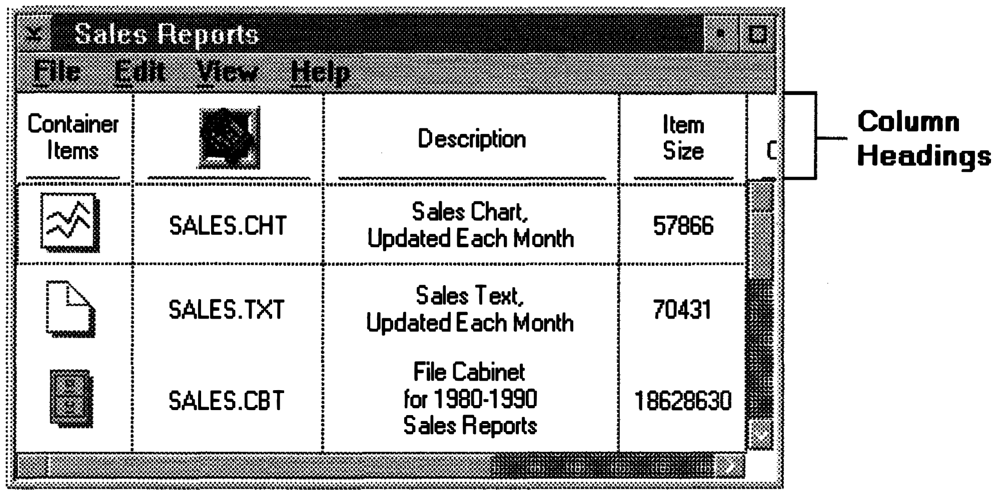


Figure 18-13. Details View

Split Bar Support for the Details View
A split bar enables the application to split the container window vertically between two column boundaries. This function is available only in the details view.

The two portions of the work area on either side of the split bar appear side-by-side. They scroll in unison vertically, but they scroll independently horizontally.

The application is responsible for specifying the position of the split bar, which is defined with the **xVertSplitbar** field. Also, the rightmost column of the left split window is specified with the **pFieldInfoLast** field. **xVertSplitbar** and **pFieldInfoLast** are fields of the CNRINFO data structure. Refer to the *OS/2 2.0 Programming Reference* for a description of the CNRINFO data structure.

The left split window cannot be empty if there is data in the right window. The right split window is not required to have data. However, because data cannot be scrolled from the right split window into the left split window, or from left to right, the split bar loses much of its usefulness if the right split window is empty.

The user can drag the vertical split bar within the limits of the window. As the user drags the split bar to the left, the right split window becomes wider; and as the split bar is dragged to the right, the left split window becomes wider.

Each container control can have one vertical split bar. Horizontal split bars are not supported.

Figure 18-14 shows a split bar between the Description column and the Date Created column.

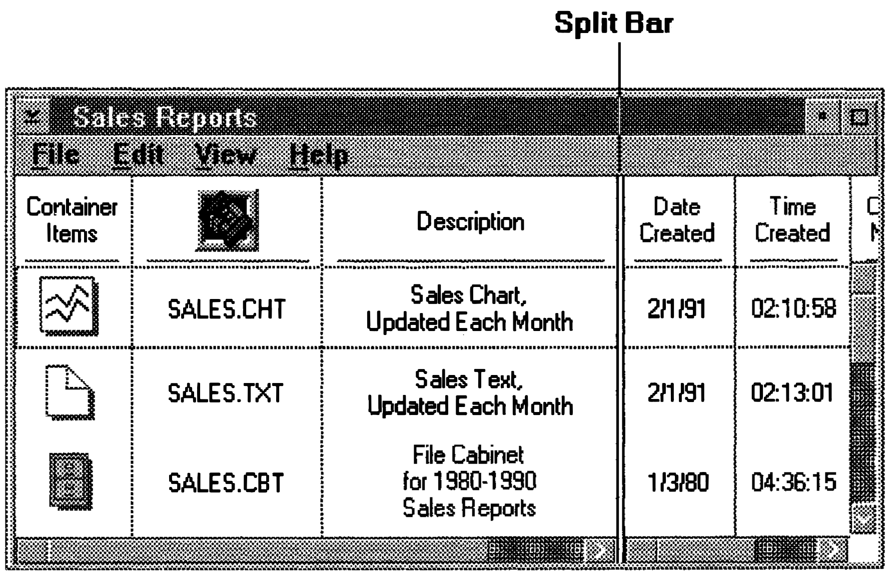


Figure 18-14. Details View with Split Bar

Changing a Container View

The sample code in Figure 18-15 shows how to use the CM_SETCNRINFO message to change from the current view of a container (name, details, or text) to the icon view.

```
CNRINFO cnrInfo;

/**********************************************************************/
/* Set the attribute field to the icon view.                          */
/**********************************************************************/
cnrInfo.flWindowAttr = CV_ICON;

/**********************************************************************/
/* Change the view from the current view to the icon view.            */
/**********************************************************************/
WinSendMsg(
  hwndCnr,                      /* Container window handle           */
  CM_SETCNRINFO,                /* Container message for setting     */
  MPFROMP(&cnrInfo),            /* Container control data            */
  MPFROMLONG(
    CMA_FLWINDOWATTR));         /* Message attribute that sets       */
                                /* container window attributes       */
```

Figure 18-15. Sample Code for Changing a Container View

Refer to the *OS/2 2.0 Programming Reference* for a complete description of the CM_SETCNRINFO message.

Using a Container

You need the following information to use a container control in your application after it is created:

- Inserting container records
- Removing container records
- Setting the container control focus.

Inserting Container Records

After the memory is allocated (see "Allocating Memory for Container Records" on page 18-4), you can insert one or more container records by using the CM_INSERTRECORD message. This message enables you to provide two pointers. The first pointer points to the record or records that are to be inserted, which is specified in the **pRecord** parameter. When you are inserting multiple records, use this parameter to specify a pointer to a linked list of records.

The second pointer points to a RECORDINSERT data structure, which specifies information the container needs for inserting records.

One of the elements of information that this data structure contains is the order in which the record or records are to be inserted, which is specified in the **pRecordOrder** field. In this field you have two options. The first option is to specify a pointer to a container record. The record or records being inserted will be placed immediately after that record. In this case, the **pRecordParent** field is ignored.

The second option is to specify whether the record or records being inserted are to be placed at the beginning or end of a list of records. This is done by specifying either the CMA_FIRST or CMA_END attributes, respectively. If you choose this option, the list of records used depends on the value of the **pRecordParent** field.

If CMA_FIRST or CMA_END is specified and the value of the **pRecordParent** field is NULL, the inserted record or records are placed at the beginning or end, respectively, of the root level records. However, if CMA_FIRST or CMA_END is specified and **pRecordParent** contains a pointer to a parent item record, the records are inserted at the beginning or end, respectively, of the list of child item records that this parent record contains. See "Tree View" on page 18-10 for more information about root level, parent, and child items.

The RECORDINSERT structure also lets you specify the z-order position of the record or records being inserted. The CMA_TOP and CMA_BOTTOM attributes of the **zOrder** field place the record at the top or bottom, respectively, relative to the other records in the z-order list. This field applies to the icon view only.

To specify the number of records that are being inserted, use the **cRecordsInsert** field. The value of this field must be greater than 0.

The last field in the RECORDINSERT structure is **fInvalidateRecord**, which enables you to control whether the record or records are displayed automatically when they are inserted. If you specify TRUE in this field, the display is updated automatically. However, if you specify FALSE, the application must send the CM_INVALIDATERECORD message after the record or records are inserted to update the display.

Where items are positioned in a container depends on the view the user has specified. If the icon view is specified and the CCS_AUTOPOSITION style bit is not set, the *x*- and *y*-coordinates for each record, which are stored in the **ptlIcon** field of the RECORDCORE and MINIRECORDCORE data structures, determine its position. Records displayed in the name view, text view, tree view, and details view are positioned as previously described in this section.

Note: Records inserted into a list of child record items can be displayed in the tree view only. These records will be visible only if the parent record item to which these child record items belong is expanded.

Figure 18-16 provides sample code that inserts a record into a container.

```
typedef struct _USERRECORD {
   RECORDCORE   RecordCore;       /* RECORDCORE structure              */
   PSZ          pszFile;          /* Text string for the details view  */
                                  /* column                            */
   CDATE        Date;             /* Date for details view column      */
}  USERRECORD, *PUSERRECORD;      /* User-defined record declaration   */


HWND             hwndCnr;         /* Container window handle           */
PUSERRECORD      pUserRecord;     /* Pointer to user-defined data      */
                                  /* structure                         */
ULONG            nRecords = 1;    /* Number of records                 */
ULONG            cbRecordData;    /* Number of bytes of additional     */
                                  /* memory                            */
RECORDINSERT     recordInsert;    /* RECORDINSERT structure            */
HPS              hps;             /* Handle to a presentation space    */
HMODULE          hmodIcons;       /* Module handle for resources       */


hps = WinGetPS (hwndCnr);
DosLoadModule ("Container Tester", 16, "ICONS", &hmodIcons);


/**********************************************************************/
/* Allocate memory for USERRECORD                                     */
/**********************************************************************/
cbRecordData = (ULONG)(sizeof(USERRECORD) - sizeof(RECORDCORE));


pUserRecord = (PUSERRECORD)WinSendMsg(
               hwndCnr,
               CM_ALLOCRECORD,
               MPFROMLONG(cbRecordData),
               (MPARAM)nRecords);


/**********************************************************************/
/* Allocate memory for text strings, as application requires.         */
/**********************************************************************/

/**********************************************************************/
/* Initialize text strings.                                           */
/**********************************************************************/
strcpy  (pUserRecord->RecordCore.pszText, "Text View");
strcpy  (pUserRecord->RecordCore.pszName, "Name View");
strcpy  (pUserRecord->RecordCore.pszIcon, "Icon View");
strcpy  (pUserRecord->RecordCore.pszTree, "Tree View");
strcpy  (pUserRecord->pszFile, "File Name");
```

Figure 18-16 (Part 1 of 2). Sample Code for Inserting a Record into a Container

```
/**********************************************************************/
/* Initialize date.                                                   */
/**********************************************************************/
pUserRecord->Date.day = 5;
pUserRecord->Date.month = 2;
pUserRecord->Date.year = 91;

/**********************************************************************/
/* Initialize attributes, icon, and bit-map pointers.                 */
/**********************************************************************/
pUserRecord->RecordCore.hptrIcon = WinLoadPointer(HWND_DESKTOP,
                                                  hmodIcons,
                                                  5000);


pUserRecord->RecordCore.hbmBitmap = GpiLoadBitmap(hps,
                                                  hmodIcons,
                                                  8000,
                                                  0L, 0L);
/**********************************************************************/
/* Initialize the record position for the icon view.                  */
/**********************************************************************/
pUserRecord->RecordCore.ptlIcon.x = 100;
pUserRecord->RecordCore.ptlIcon.y = 200;

/**********************************************************************/
/* Initialize CM_INSERTRECORD data structure.                         */
/**********************************************************************/
recordInsert.pRecordOrder = (PRECORDCORE)CMA_FIRST;
recordInsert.zOrder = (ULONG)CMA_TOP;
recordInsert.cRecordsInsert = nRecords;
recordInsert.fInvalidateRecord = TRUE;
recordInsert.pRecordParent = NULL;


/**********************************************************************/
/* Insert record.                                                     */
/**********************************************************************/
WinSendMsg(hwndCnr,
           CM_INSERTRECORD,
           MPFROMP(pUserRecord),
           MPFROMP(&recordInsert));


/**********************************************************************/
/* Clean up.                                                          */
/**********************************************************************/
DosFreeModule(hmodIcons);
WinReleasePS(hps);
```

Figure 18-16 (Part 2 of 2). Sample Code for Inserting a Record into a Container

Removing Container Records

The CM_REMOVERECORD message can be used to remove one or more container records from the container control. The application must set the pointers to each record in an array to be removed.

If the **fRemoveRecord** parameter of this message includes the CMA_FREE attribute, the records are removed and the memory is freed. If this attribute is not set, the records are removed from the list of items in the container, and the application must use the CM_FREERECORD message to free the memory. The default is to not free the memory.

If the **fRemoveRecord** parameter includes the CMA_INVALIDATERECORD attribute, the container is invalidated after the records are removed. The default is to not invalidate the container. The CMA_INVALIDATERECORD attribute can be used with the CMA_FREE attribute, separated by a logical OR operator (|), to free the record's memory and invalidate the container.

The sample code in Figure 18-17 removes all records from a container and frees the memory associated with those records. It is the application's responsibility to free all application-allocated memory that is associated with the removed container records. The container is invalidated and repainted.

```
USHORT cNumRecord;              /* Number of records to be removed     */
USHORT fRemoveRecord;           /* Container message attributes        */

/**********************************************************************/
/* Zero means remove all records.                                     */
/**********************************************************************/
cNumRecord = 0;

/**********************************************************************/
/* Specify attributes to invalidate the container and free the memory */
/**********************************************************************/
fRemoveRecord =
  CMA_INVALIDATERECORD | CMA_FREE;

/**********************************************************************/
/* Remove the records.                                                */
/**********************************************************************/
WinSendMsg(hwndCnr,             /* Container window handle            */
  CM_REMOVERECORD,              /* Container message for removing     */
                                /* records                            */
  NULL,                         /* NULL PRECORDARRAY                  */
  MPFROM2SHORT(
    cNumRecord,                 /* Number of records                  */
    fRemoveRecord));            /* Memory invalidation flags          */
```

Figure 18-17. Sample Code for Removing Container Records

Setting the Container Control Focus

The application must set the focus of the container control by using the WinSetFocus function.

Graphical User Interface Support

The following describes the container control support for graphical user interfaces (GUIs). Except where noted, this support conforms to the guidelines in the *SAA CUA Advanced Interface Design Reference.* The GUI support provided by the container control consists of:

- Scrolling
- Selecting container items
- Providing emphasis
- Using direct manipulation
- Specifying space between container items.

Scrolling

The container control automatically provides horizontal or vertical scroll bars, or both, whenever all or part of one or more container items are not visible in a container window's work area.

If all container items are visible in the work area, the scroll bars are either removed or disabled, depending on the view and how the items are positioned, as follows:

- If container items are displayed in the icon or tree view, and one or more items are not visible in the work area, a horizontal scroll bar, vertical scroll bar, or both, are provided, depending on the position of the items outside of the work area. If container items are positioned to the right or left of the work area, a horizontal scroll bar is provided; if container items are positioned below or above the work area, a vertical scroll bar is provided.

  Scroll bars are not provided if all the container items are visible in the work area. Scroll bars are removed from the container window if either of the following occurs:

  - Container items positioned outside the work area are moved into the work area
  - The size of the container window is increased so that container items formerly not visible become visible.

- If container items are displayed in non-flowed text and non-flowed name views, a vertical scroll bar is provided; this scroll bar is disabled if all the container items are visible in the work area. A horizontal scroll bar is used in these views only when the work area is too narrow to allow the widest container item to be seen in its entirety. If the user changes the window size to allow the entire widest container item to be seen, the horizontal scroll bar is removed.

- If container items are displayed in flowed text and flowed name views, a horizontal scroll bar is provided; this scroll bar is disabled if all the container items are visible in the work area. A vertical scroll bar is used in these views only when the work area is too short to allow the tallest container item to be seen in its entirety. If the user changes the window size to allow the entire tallest container item or items to be seen, the vertical scroll bar is removed.

- If container items are displayed in the details view, both horizontal and vertical scroll bars are provided. These scroll bars are disabled if all the container items are visible in the work area.

  Note: A details view that is split has two horizontal scroll bars, one for each portion of the split window.

Dynamic Scrolling

The container control supports *dynamic scrolling*, which enables the user to drag the scroll box in the scroll bar and get immediate visible feedback on where the scrolling will stop when the scroll box is dropped. If the scrolling range is greater than 32KB pels, dynamic scrolling is disabled.

Selecting Container Items

Except during direct manipulation and direct editing of text in a container, a user must select a container item before performing an action on it. The container control provides several selection types, along with selection techniques to implement those types. The container control also supports two selection mechanisms: any pointing device, such as a mouse, and the keyboard.

Selection Types

The container control supports the following selection types:

- **Single selection**

  Single selection enables a user to select only one container item at a time. This is the default selection type for all views and is the only selection type supported for the tree view.

- **Extended selection**

  Extended selection enables a user to select one or more container items, in any combination. The CUA-defined keyboard augmentation keys are implemented for extended selection. When used with a pointing device, these keys enable a user to select discontiguous sets of container items. Extended selection is valid for all views except the tree view.

- **Multiple selection**

  Multiple selection enables a user to select none, some, or all of the container items. Multiple selection is valid for all views except the tree view.

Only one of these selection types can be used for each container. The selection type for a container is defined when the container is created.

These selection types conform to the guidelines in the *SAA CUA Advanced Interface Design Reference*. Refer to that book for detailed information.

Selection Techniques

Depending on the type of view and the type of selection, a user can select container items using the following selection techniques:

- **Marquee selection**

  Marquee selection is supported only in the icon view and is only valid with the extended and multiple selection types. This selection technique enables a user to begin selection from an anchor point that is established by moving the pointer to white space in the container and pressing, but not releasing, the select button on the pointing device. As the user presses the select button and drags the

pointer, a tracking rectangle is drawn between the anchor point and the current pointer position. All items whose icons or bit maps are entirely within the tracking rectangle are dynamically selected.

- **Swipe selection**

  Swipe selection is valid only with the extended and multiple selection types. The container control implements two techniques for swipe selection: touch swipe and range swipe.

  - **Touch swipe**

    Touch swipe selection is implemented in the icon view. With this selection technique, the pointer must pass over some portion of a container item while the user is pressing the select button for that item to be selected.

  - **Range swipe**

    In views other than the icon and tree views, range swipe selection is available. With this method, the user presses the select button while moving the pointer. However, the pointer does not have to pass directly over a container item for that item to be selected. Aside from pressing the select button and moving the pointer, the only other requirement for selection is that the container item must be within a range of items that is being selected. The range begins at the pointer's position when the user presses the select button; it ends at the pointer's position when the user releases the select button. Refer to the *SAA CUA Advanced Interface Design Reference* for complete information on touch swipe and range swipe selection.

- **First-letter selection**

  For the icon, name, text, and tree views, first letter selection occurs when a character key is pressed, and the first container item whose text begins with that character is displayed with selected-state emphasis. The same is true for the details view, except that all the columns for a record are searched for a matching character before the next record is searched. The effect of first letter selection on other selected container items depends on the chosen selection type (single, multiple, or extended).

All these selection techniques conform to the descriptions in the *SAA CUA Guide to User Interface Design*.

Note: If more than one container window is open, selecting a container item in one window has no effect on the selections in any other window.

Selection Mechanisms

The *SAA CUA Guide to User Interface Design* defines mouse button 1, the select button, to be used for selecting container items and mouse button 2, the drag button, to be used for dragging and dropping container items during direct manipulation. These definitions also apply to the same buttons on any other pointing device.

In addition, a user can press a keyboard key while pressing a mouse button; this is called *keyboard augmentation*. The only instance of keyboard augmentation defined specifically for the container control is pressing the Alt key with the select button, which starts direct editing of text in a container. Refer to the *SAA CUA Advanced Interface Design Reference* for a complete list of the keys that are defined in the CUA guidelines for keyboard augmentation.

In addition, the container control supports two keyboard cursors that can be moved by using keyboard navigation keys:

- The *selection cursor*, a dotted black box drawn around a container item, which represents the current position for the purpose of keyboard navigation.
- The *text cursor*, a vertical line that shows the user where text can be inserted or deleted when container text is being edited directly.

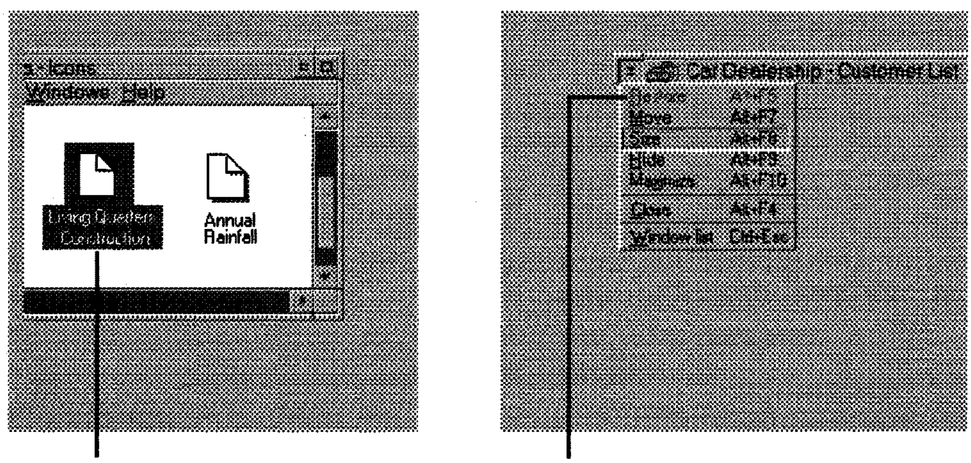
Keyboard navigation consists of the use of the Up, Down, Left, and Right Arrow keys, the Home key, the End key, the PgUp (page up) key, and the PgDn (page down) key. If container items are not visible within the work area, navigation with these keys causes the items to scroll into view if the user is not editing container text directly. Refer to the *SAA CUA Guide to User Interface Design* for a description of the keyboard interface model.

Providing Emphasis

The container control supports various types of emphasis. Emphasis is applied as described in the *SAA CUA Guide to User Interface Design*. Refer to that book for complete information about the use of emphasis. The following list describes forms of emphasis that have a distinct visible representation in the container control:

- **Selected-state emphasis**

  When a container item is selected, the entire container item receives *selected-state emphasis*, which means that selected-state emphasis is applied to icon/text or bit-map/text pairs in the icon, name, tree icon, and tree name views; text strings in the text and tree text views; and an entire row that represents a container item in the details view. Figure 18-18 illustrates selected-state and unavailable-state emphasis; the emphasis on the choice in the pull-down menu indicates that the choice is unavailable.



Selected-state emphasis **Unavailable-state emphasis**

Figure 18-18. Selected-State and Unavailable-State Emphasis

  The color for selected-state emphasis can be changed by using the control panel, or the WinSetPresParam function, which results in a WM_PRESPARAMCHANGED message being sent to the container. See the WinSetPresParam function and WM_PRESPARAMCHANGED (in Container Controls) message in the *OS/2 2.0 Programming Reference* for more information.

- **In-use emphasis**

  Cross-hatching behind an icon or bit map indicates *in-use emphasis*. In-use emphasis is not applied to container items in the text view, tree text view, or details view when it contains text only. However, the details view often includes icons or bit maps in one column of each record, usually the leftmost column. In this situation, specify the column that contains the icons or bit maps so that in-use emphasis can be applied to them. This column can be set by using the **pFieldInfoObject** field of the CNRINFO data structure.

- **Target emphasis**

  Target emphasis is used during direct manipulation. When a user drags one container item over another, the item beneath the dragged item displays *target emphasis*. Two forms of target emphasis (visible feedback) are available: a black line and a black border. These forms of emphasis indicate the *target*, where the container item will be dropped if the user releases the drag button. The CA_ORDEREDTARGETEMPH and CA_MIXEDTARGETEMPH attributes of the CNRINFO data structure's **flWindowAttr** field determine the form of emphasis applied for the text, name, and details views, as follows:

  – If the CA_ORDEREDTARGETEMPH attribute is set:

    - The CN_DRAGAFTER notification code is sent when a container item is being dragged.

    - A black line is drawn between container items to show the current target position.

  – If the CA_MIXEDTARGETEMPH attribute is set:

    - The CN_DRAGAFTER and CN_DRAGOVER notification codes are sent when a container item is being dragged. The notification code sent depends on the position of the pointer relative to the item it is positioned over.

    - A black line is drawn if the pointer is positioned such that the item being dragged will be inserted between two target items.

    - A black border is drawn around either the entire target item for the text and details views or the icon or bit map for the name view if the pointer is positioned such that the item being dragged will be dropped on the target item.

  – If the CA_ORDEREDTARGETEMPH and CA_MIXEDTARGETEMPH attributes are not set:

    - The CN_DRAGOVER notification code is sent when a container item is being dragged.

    - A black border is drawn around the entire target item for the text and details views, and around the icon or bit map only for the name view.

For the icon and tree view, the CA_ORDEREDTARGETEMPH and CA_MIXEDTARGETEMPH attributes are ignored, so target emphasis is applied as follows:

- The CN_DRAGOVER notification code is sent when a container item is dragged.
- A black border is drawn around the target, as follows:
  - For the icon view, if the target is another container item, a black border is drawn around the icon or bit map that represents the container item, but not around the text string beneath it. If the target is white space, a black border is drawn around the outer edge of the entire work area.
  - For the tree icon and tree name views, a black border is drawn around the icon or bit map that represents the container item, but not around the text string to the right of it.
  - For the tree text view, a black border is drawn around the entire target item.

Using Direct Manipulation

Direct manipulation is a protocol that enables the user to drag a container item within its current window or from one window to another. The user can drop the container item either on white space in a window or on another item.

Direct manipulation can be performed with all views of the container control. An API is provided so that the application is notified if an item is dropped on another item in the container and if an item is dragged from the container.

The user can drag any container item, whether or not it is selected. If the user presses the drag button when the pointer is over a selected container item, the application drags all selected items. See “Selection Techniques” on page 18-23 for information about the selection techniques.

If the user presses the drag button when the pointer is over a container item that is not selected, the application drags only the item that the pointer is over.

The container control fully supports direct manipulation. Refer to the *SAA CUA Guide to User Interface Design* for more information about the effects of direct manipulation.

Specifying Space between Container Items

You can specify the amount of vertical space, in pels, to allow between container items by using the **cyLineSpacing** field of the CNRINFO data structure. If you do not specify how much vertical space can be used, the container control sets the space between the items using a default value. For the tree view, you can specify the horizontal distance between the levels by using the **cxTreeIndent** field of the CNRINFO data structure. If this value is less than 0, a default is used.

Enhancing Container Control Performance

The following offers information about fine-tuning a container to enhance its performance and effectiveness:

- Positioning container items
- Specifying deltas for large amounts of data
- Direct editing of text in a container
- Specifying container titles
- Specifying fonts and colors
- Drawing container items and painting backgrounds
- Filtering container items
- Optimizing container memory usage
- Sharing records among multiple containers.

Positioning Container Items

Container items are positioned in the icon view according to workspace coordinates.

The *workspace* is a two-dimensional Cartesian coordinate system. The user can see a portion of the workspace in the work area, which is the scrollable viewing area of the container that is defined by the size of the container window. The work area is logically scrollable within the workspace.

Figure 18-19 on page 18-29 shows the *x*- and *y*-axes of the workspace with a container window and its work area superimposed. (This figure is not drawn to scale.)

Scrollable Workspace Areas

Figure 18-19 on page 18-29 shows the scrollable area of the workspace, and thus the container.

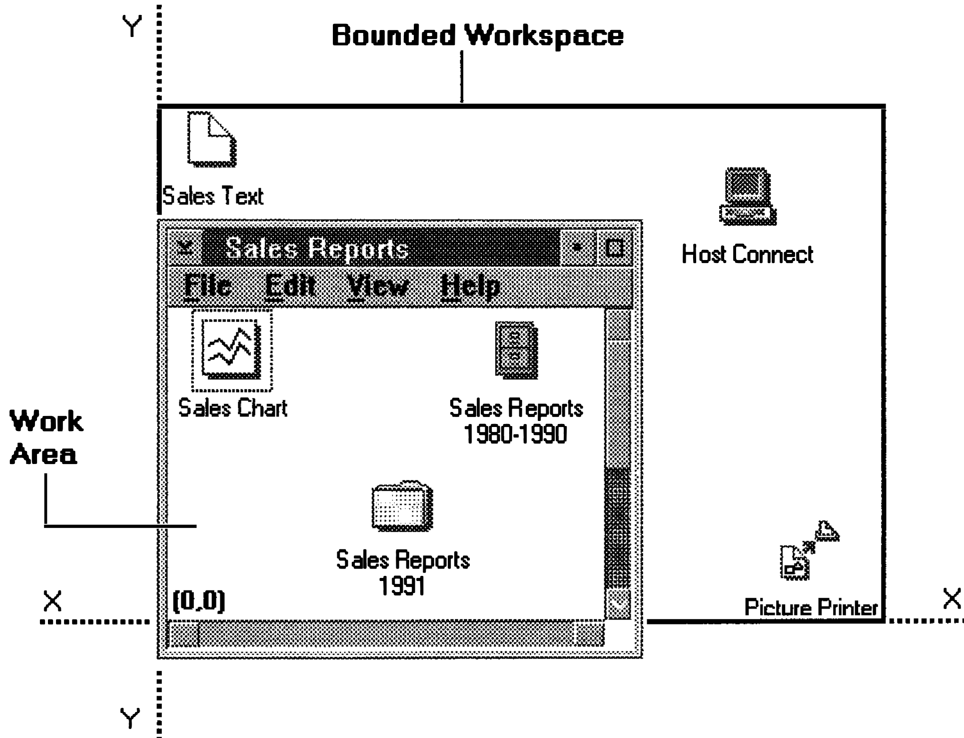


Figure 18-19. Workspace X- and Y-Axes

This area is indicated by the solid black line that runs even with:

- The top and bottom edges, respectively, of the topmost and bottommost container items
- The left and right edges, respectively, of the leftmost and rightmost container items.

The scrollable workspace area, then, is defined by the minimum and maximum *x*- and *y*-coordinates of the items in the container. That is, the work area of the container window can be scrolled only within the workspace and only as far as is necessary to see the topmost, bottommost, leftmost, and rightmost container items.

Figure 18-20 further illustrates a bounded workspace. In this example, the topmost and bottommost container items limit the workspace.

In Figure 18-20, the work area has been scrolled so that all elements are not within the work area. The work area could be scrolled to the left so that it would include the leftmost element, or scrolled down and to the right to include the rightmost element, but it could not be scrolled any farther in either direction.

Workspace and Work Area Origins

When the container is created, the work area and workspace share the same origin, **(0,0)**, as represented in Figure 18-19 on page 18-29. If the application requires that the work area and the workspace have different origins, the application can use the **ptlOrigin** field of the CNRINFO data structure and the CM_SETCNRINFO message to set the origin of the work area. The application could use the CM_QUERYCNRINFO and CM_SETCNRINFO messages to obtain the origin when the user ends the application, and reset it when the user restarts the application.

Container items are located in reference to the workspace origin. There is a visual shift as the work area is scrolled; but because the work area moves over a fixed workspace, the coordinates of the container items do not change.

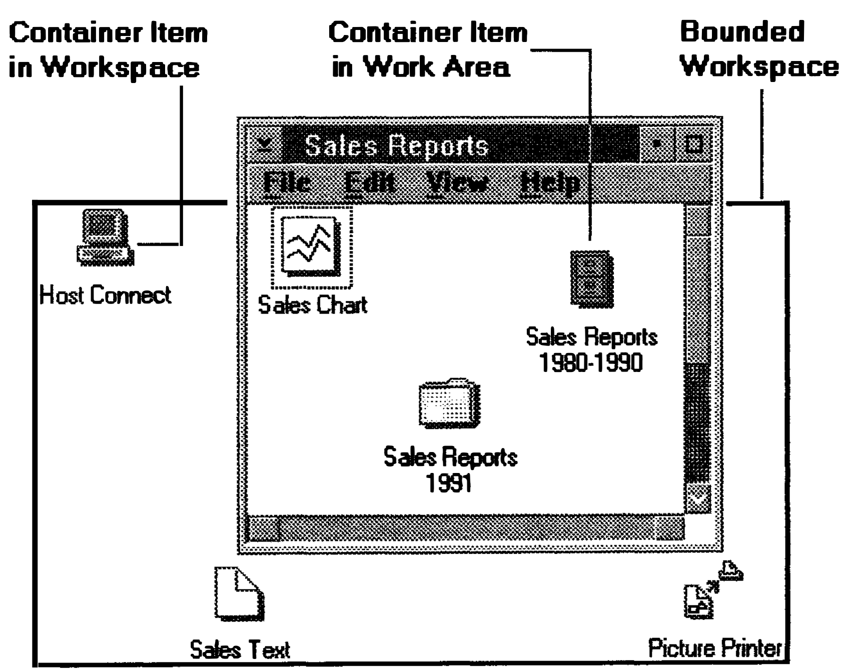


Figure 18-20. Workspace Bounds

Specifying Deltas for Large Amounts of Data

The container control can accommodate large amounts of data with an application-defined delta. The *delta* is an application-defined threshold, or number of container items, from either end of the list. The application is responsible for specifying the delta value in the CNRINFO data structure's **cDelta** field. It also is responsible for setting the delta value with the CMA_DELTA attribute of the CM_SETCNRINFO message's **ulCnrInfoFl** parameter. Refer to the *OS/2 2.0 Programming Reference* for description of CM_SETCNRINFO message.

The container control monitors its place in the list of container items when the user is scrolling through it. When the user scrolls to the delta from either end of the list, the container control sends a CN_QUERYDELTA notification code to the application as a request for more container items in the list.

The application is responsible for managing the records in the container. When the application receives the CN_QUERYDELTA notification code, the application is responsible for removing and inserting container records by using the CM_REMOVERECORD message and the CM_INSERTRECORD message respectively.

Notes:

1. The delta concept is intended for applications with large amounts of data, or several thousand records. Applications with smaller amounts of data are not required to use the delta function. The default delta value is 0.
2. The delta function is not available in the icon view because it is intended for data displayed in a linear format.

Direct Editing of Text in a Container

Direct editing of text is supported for any text field in a container, including the container title, column headings, and container items. If a text field, such as the text field beneath an icon in the icon view, has no text and is not read-only, a user can place text in that field by editing the field directly. The font specified for the container by the application is used for the edited text.

Direct editing is supported only for text data. Therefore, if the data type in the details view is other than CFA_STRING, a user cannot edit it. CFA_STRING is an attribute of the FIELDINFO data structure's **flData** field.

You can prevent a user from editing any of the text in a container window by setting the CCS_READONLY style bit when a container is created. If you do not set this style bit, the user can edit any of the text in a container window unless you set the following read-only attributes: CA_TITLEREADONLY, CRA_RECORDREADONLY, CFA_FIREADONLY, and CFA_FITITLEREADONLY. If a read-only attribute is set, a user's attempts to edit container text directly are ignored. See the description of the CCS_READONLY style bit in the *OS/2 2.0 Programming Reference* for more information about these attributes.

A user can edit container text directly by doing either of the following:

- Moving the pointer to an editable text field, holding down the Alt key, and clicking the select button
- Sending a CM_OPENEDIT message to the container control.

  The application can assign a key or menu choice to this message so that the keyboard can be used to edit container text directly.

The container control responds by using the WM_CONTROL message to send the CN_BEGINEDIT notification code to the application. A window that contains a multiple-line entry (MLE) field opens to show that container text can be edited directly.

The editing actions supported by MLEs, such as Cut, Copy, and Paste, are also supported by the container control. These actions can be performed using system-defined *shortcut* keys. The actions and shortcut keys are defined in the *SAA CUA Advanced Interface Design Reference*.

If the user enters a text string that is longer than the text field, the text string scrolls. Also, if multiple lines of text are wanted, a user can press the Enter key and type on the following line whenever another line is needed.

A user can end the direct editing of container text and save the changes by doing either of the following:

- Moving the pointer outside the MLE and pressing the select button
- Sending a CM_CLOSEEDIT message to the container control.

  The application can assign a key or menu choice to this message so that the keyboard can be used to end the direct editing of container text.

The container responds by sending the WM_CONTROL message to the application again, but this time with the CN_REALLOCPSZ notification code. The application can allocate more memory on receipt of the CN_REALLOCPSZ notification code, if necessary. If the application returns TRUE, the container control copies the new text to the application's text string. If the application returns FALSE, the text change in the MLE is disregarded. The container then sends the WM_CONTROL message to the application again, this time with the CN_ENDEDIT notification code. The MLE field is removed from the screen, leaving only the text string.

A user can end the direct editing of container text without saving any changes to the text in numerous ways, including the following:

- Pressing the Esc key
- Dragging the container item that is being edited
- Pressing the Alt key and the select button before the direct editing of container text has ended
- Scrolling the container window.

The CN_ENDEDIT notification code is sent to the application in each of these cases.

Specifying Container Titles

The container control can have a non-scrollable title that consists of one or more lines of text. The container control does not limit the number of lines or the number of characters in each line. If specified, this title is the first line or lines of the container control. The text of the title is determined by the application and can be

used to identify the container or to contain status information. Figure 18-21 on page 18-33 shows an example of a container title.

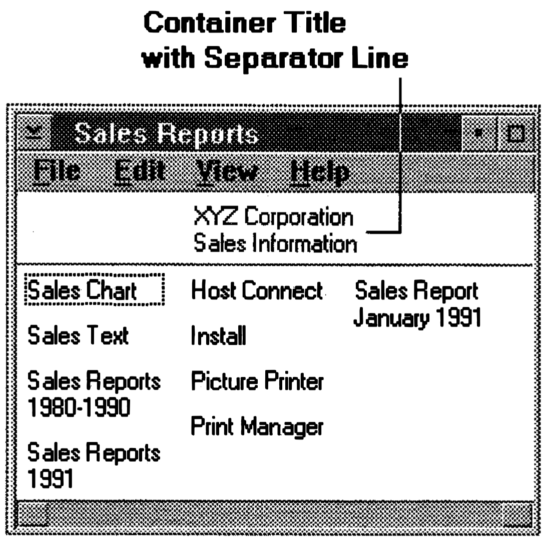


Figure 18-21. Non-Flowed Text View with Container Title

The CA_CONTAINERTITLE attribute must be set to include a title in a container window. The default is no container title.

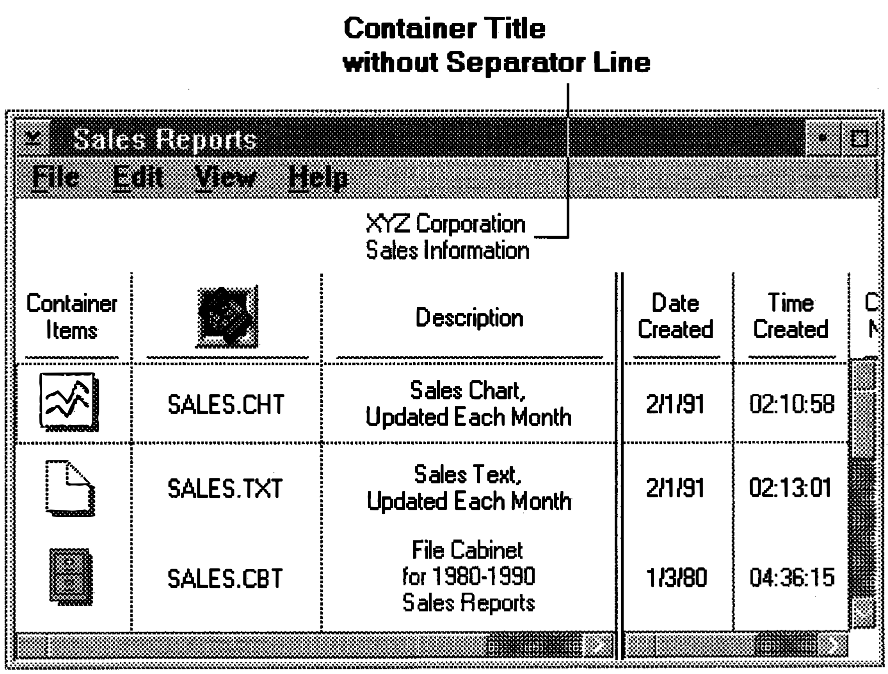


Figure 18-22. Split Details View with Container Title

If you do not want the user to be able to edit the container title directly, you can set the CA_TITLEREADONLY attribute. The default is that the container title can be edited. See "Direct Editing of Text in a Container" on page 18-31 for more information about editing container text directly.

Below the title in Figure 18-21, a horizontal line separates the container title from the container items. The CA_TITLESEPARATOR attribute must be set in order to

include a separator line in a container window. The default is no separator line, as shown in Figure 18-22.

The container titles in both figures are centered. This is the default. However, the CA_TITLECENTER, CA_TITLELEFT, or CA_TITLERIGHT attribute can be used to specify whether a container title is to be centered, left-justified, or right-justified, respectively.

All the container attributes described here are attributes of the CNRINFO data structure's **flWindowAttr** field.

Specifying Fonts and Colors

A different font can be specified for each view. The same font is used for the text within each view. Text color can be configured from the system control panel. The application can override the system-defined font and colors by using the WinSetPresParam function.

The font and color can be changed for the text in all views. However, font and color cannot be changed for text in individual columns in the details view. Therefore, all text in the details view, including the container title, columns, and column headings, has the same font and color.

Drawing Container Items and Painting Backgrounds

The container control enables your application to paint the container's background, draw the container items, or both. If the CA_OWNERPAINTBACKGROUND attribute is set, the container control sends the CM_PAINTBACKGROUND message to itself. Your application can control background painting by subclassing the container control and intercepting the CM_PAINTBACKGROUND message. CA_OWNERPAINTBACKGROUND is an attribute of the CNRINFO data structure's **flWindowAttr** field.

To support *ownerdraw*, the drawing of container items by the application, the container control provides the CA_OWNERDRAW attribute of the CNRINFO data structure's **flWindowAttr** field. If this attribute is set and the application processes the WM_DRAWITEM window message, the application is responsible for drawing each container item, including the types of emphasis.

In addition, the container control supports ownerdraw for each column in the details view. This support is indicated by the CFA_OWNER attribute, which is specified in the FIELDINFO data structure's **flData** field.

If the CA_OWNERDRAW attribute or CFA_OWNER attribute is set, the container control sends the application a WM_DRAWITEM message with a pointer to an OWNERITEM data structure as the **ownerItem** parameter. Refer to the *OS/2 2.0 Programming Reference* for a description of the *OWNERITEM* data structure fields as they apply to the container control.

Filtering Container Items

If the CRA_FILTERED attribute is set for a container item, that item is not displayed. Therefore, filtering can be used to hide container items. CRA_FILTERED is an attribute of the RECORDCORE data structure's **flRecordAttr** field.

Optimizing Container Memory Usage

The container control provides an option to enable you to develop applications that minimize the amount of memory used for each container record. This is done by specifying the CCS_MINIRECORDCORE style bit when the container is created, which causes a smaller version of the RECORDCORE data structure, MINIRECORDCORE, to be used. The following table shows the differences between these two data structures.

Table 18-3. Differences between RECORDCORE and MINIRECORDCORE

RECORDCORE	MINIRECORDCORE
Up to eight image handles can be specified for each record.	Only one image handle can be specified for each record. **Note:** This image must be an icon.
Up to four text strings can be specified for each record.	Only one text string can be specified for each record.

Allocating Memory for Container Records When Using MINIRECORDCORE

The sample code in Figure 18-23 shows how to allocate memory for one container record when the MINIRECORDCORE data structure is used. A pointer to the MINIRECORDCORE structure is returned. This is the same sample code as that used in Figure 18-2 on page 18-4 except for one line, which is highlighted.

```
HWND        hwndCnr;              /* Container window handle           */
PRECORDCORE pRecord;              /* Pointer to RECORDCORE structure   */
ULONG       nRecords = 1;         /* 1 record to be allocated          */

pRecord =
  (PMINIRECORDCORE)WinSendMsg(
    hwndCnr,                      /* Container window handle           */
    CM_ALLOCRECORD,               /* Message for allocating the record */
    NULL,                         /* No additional memory              */
    (MPARAM)nRecords);            /* Number of records to be allocated */
```

Figure 18-23. Sample Code for Allocating Memory for Smaller Container Records

Sharing Records Among Multiple Containers

The container control enables the application to share records that are allocated among multiple containers in the same process. That is, records can be allocated once and then inserted into many containers in the same process. Only one copy of each record is in memory, but the container provides the flexibility for the records to appear as though they are independent of one another.

When a record is inserted into the container, the **flRecordAttr** and **ptlIcon** fields of the record structure are saved internally. The values in these fields cause the record attributes for all views and the icon position for the icon view to be associated with the specific container into which the record is inserted. If the same record is inserted into multiple containers, the attributes and icon location of each record are maintained separately. The application uses the CM_QUERYRECORDINFO message to retrieve the current values of these two fields for a particular record in a specific container.

Invalidating Records Shared by Multiple Containers

When a record is invalidated by an application, the **flRecordAttr** and **ptlIcon** fields are saved internally, just as when a record is inserted. Therefore, use the CM_QUERYRECORDINFO message to acquire the current data for each record that is being invalidated. After querying the current data, you can change any of this data before invalidating its record.

Freeing Records Shared by Multiple Containers

When an application attempts to free a record in an open container, the record is freed only if it is not being used in any other open container. The methods of freeing records in an open container are to use the CM_FREERECORD message, or use the CM_REMOVERECORD message and specify the CMA_FREE attribute.

Summary

Following are tables that describe the OS/2 container control structures, notification codes, notification messages, and window messages:

Table 18-4 (Page 1 of 2). Container Control Structures

Structure name	Description
CDATE	Contains date information for a data element in the details view of the container.
CNRDRAGINFO	Contains information about a direct manipulation event occurring over the container.
CNRDRAGINIT	Contains information about a direct manipulation event that was initiated in a container.
CNRDRAWITEMINFO	Contains information about the item being drawn in the container.
CNREDITDATA	Contains information about the direct editing of container text.
CNRINFO	Contains information about the container.
CTIME	Contains time information for a data element in the details view of the container.
FIELDINFO	Contains information about column data in the details view of the container.
FIELDINFOINSERT	Contains information about the FIELDINFO structure or structures that are being inserted into the container.
MINIRECORDCORE	Contains information for container records that are smaller than those defined by the RECORDCORE structure.
NOTIFYDELTA	Contains information about the placement of delta information for the container.
NOTIFYRECORDEMPHASIS	Contains information about the emphasis applied to a container record.
NOTIFYRECORDENTER	Contains information about the input device being used with the container.
NOTIFYSCROLL	Contains information about scrolling the container window.
OWNERBACKGROUND	Contains information about painting the container window's background.

Table 18-4 (Page 2 of 2). Container Control Structures	
Structure name	**Description**
QUERYRECFROMRECT	Contains information about a container record that is bounded by a specified rectangle.
QUERYRECORDRECT	Contains information about the rectangle that bounds a specified container record.
RECORDCORE	Contains information for container records.
RECORDINSERT	Contains information about the RECORDCORE structure or structures that are being inserted into the container.
SEARCHSTRING	Contains information about the container text string that is the object of the search.
TREEITEMDESC	Contains icons and bit maps used to represent the state of an expanded or collapsed parent item in the tree name view.

Table 18-5 (Page 1 of 2). Container Control Notification Codes	
Code name	**Description**
CN_BEGINEDIT	Sent when container text is about to be edited.
CN_COLLAPSETREE	Sent when a parent item is collapsed in the tree view.
CN_CONTEXTMENU	Sent when the container receives a WM_CONTEXTMENU message.
CN_DRAGAFTER	Sent when the container receives a DM_DRAGOVER message.
CN_DRAGLEAVE	Sent when the container receives a DM_DRAGLEAVE message.
CN_DRAGOVER	Sent when the container receives a DM_DRAGOVER message.
CN_DROP	Sent when the container receives a DM_DROP message.
CN_DROPHELP	Sent when the container receives a DM_DROPHELP message.
CN_EMPHASIS	Sent when the attributes of a container record change.
CN_ENDEDIT	Sent when direct editing of the container text ends.
CN_ENTER	Sent either when the Enter key is pressed while the container window has the focus, or when the select button is double-clicked while the pointer is over the container window.
CN_EXPANDTREE	Sent when the container expands a parent item in the tree view.
CN_HELP	Sent when the container receives a WM_HELP message.
CN_INITDRAG	Sent when the drag button is pressed and the pointer is moved while over the container control.
CN_KILLFOCUS	Sent when the container is losing the focus.

Table 18-5 (Page 2 of 2). Container Control Notification Codes	
Code name	**Description**
CN_QUERYDELTA	Sent to query for more data when the user scrolls to a preset delta value.
CN_REALLOCPSZ	Sent when container text is edited (before CN_ENDEDIT is sent).
CN_SCROLL	Sent when the container window scrolls.
CN_SETFOCUS	Sent when the container receives the focus.

Table 18-6. Container Control Notification Messages	
Message	**Description**
WM_CONTROL	Occurs when the container control has a significant event to notify to its owner.
WM_CONTROLPOINTER	Sent to the container control's owner window when the pointing device pointer moves over the container window, allowing the owner to set the pointing device pointer.
WM_DRAWITEM	Sent to the owner of the container control each time an item is to be drawn.

Table 18-7 (Page 1 of 3). Container Control Window Messages	
Message	**Description**
CM_ALLOCDETAILFIELDINFO	Allocates memory for one or more FIELDINFO structures.
CM_ALLOCRECORD	Allocates memory for one or more RECORDCORE structures.
CM_ARRANGE	Arranges the container records in the icon view.
CM_CLOSEEDIT	Closes the window containing the multiple-line entry (MLE) field used to edit container text directly.
CM_COLLAPSETREE	Causes one parent item in the tree view to be collapsed.
CM_ERASERECORD	Erases the source record from the current view when a move occurs as a result of direct manipulation.
CM_EXPANDTREE	Causes one parent item in the tree view to be expanded.
CM_FILTER	Filters the contents of a container so that a subset of the container items can be viewed.
CM_FREEDETAILFIELDINFO	Frees the memory associated with one or more FIELDINFO structures.
CM_FREERECORD	Frees the memory associated with one or more RECORDCORE structures.
CM_HORZSCROLLSPLITWINDOW	Scrolls a split window in the split details view.

Table 18-7 (Page 2 of 3). Container Control Window Messages	
Message	**Description**
CM_INSERTDETAILFIELDINFO	Inserts one or more FIELDINFO structures into a container control.
CM_INSERTRECORD	Inserts one or more RECORDCORE structures into a container control.
CM_INVALIDATEDETAILFIELDINFO	Notifies the container control that any or all FIELDINFO structures are not valid and that the view must be refreshed.
CM_INVALIDATERECORD	Notifies the container control that any or all RECORDCORE structures are not valid and must be refreshed.
CM_OPENEDIT	Opens the window that contains the multiple-line entry (MLE) field used to edit container text directly.
CM_PAINTBACKGROUND	Informs an application when a container's background is painted if the CA_OWNERPAINTBACKGROUND attribute of the CNRINFO data structure is specified.
CM_QUERYCNRINFO	Returns the container's CNRINFO structure.
CM_QUERYDETAILFIELDINFO	Returns a pointer to the requested FIELDINFO structure.
CM_QUERYDRAGIMAGE	Returns a handle to the icon or bit map for the record in the current view.
CM_QUERYRECORD	Returns a pointer to the requested RECORDCORE structure.
CM_QUERYRECORDEMPHASIS	Queries for a container record with the specified emphasis attributes.
CM_QUERYRECORDFROMRECT	Queries for a container record that is bounded by the specified rectangle.
CM_QUERYRECORDINFO	Updates the specified records with the current information for the container.
CM_QUERYRECORDRECT	Returns the rectangle of the specified container record, relative to the container window origin.
CM_QUERYVIEWPORTRECT	Returns a rectangle that contains the coordinates of the container's work area.
CM_REMOVEDETAILFIELDINFO	Removes one, multiple, or all FIELDINFO structures from the container control.
CM_REMOVERECORD	Removes one, multiple, or all RECORDCORE structures from the container control.
CM_SCROLLWINDOW	Scrolls an entire container window.
CM_SEARCHSTRING	Returns the pointer to a container record whose text matches the string.
CM_SETCNRINFO	Sets or changes the data for the container control.
CM_SETRECORDEMPHASIS	Sets the emphasis attributes of the specified container record.

Table 18-7 (Page 3 of 3). Container Control Window Messages	
Message	**Description**
CM_SORTRECORD	Sorts the container records in the container control.
WM_PRESPARAMCHANGED	Sent when a presentation parameter is set or removed dynamically from a window instance.

Chapter 19. Notebook Controls

A notebook control (WC_NOTEBOOK window class) is a visual component that organizes information on individual *pages* so that a user can find and display that information quickly and easily. This chapter explains how to use notebook controls in PM applications.

About Notebook Controls

This notebook control component simulates a real notebook, but improves on it by overcoming its natural limitations. A user can select and display pages by using a pointing device or the keyboard.

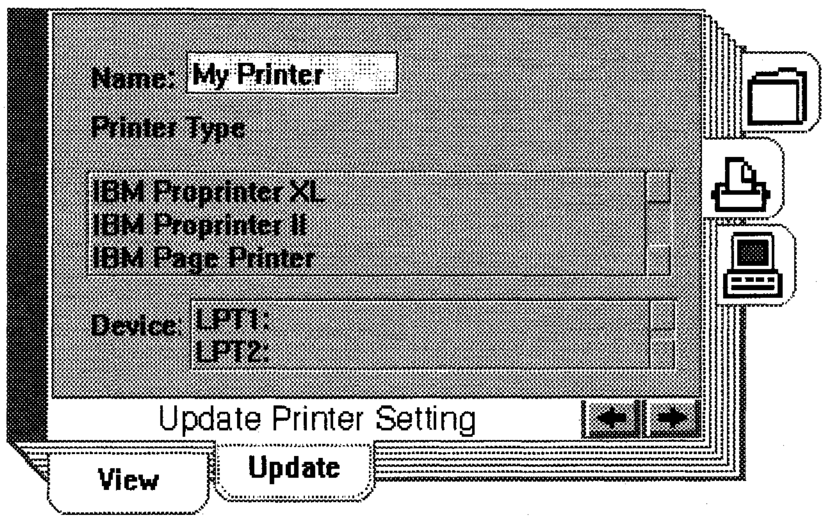


Figure 19-1. Notebook Example

The notebook can be customized to meet varying application requirements, while providing a user interface component that can be used easily to develop products that conform to the Common User Access (CUA) user interface guidelines. The application can specify different colors, sizes, and orientations for its notebooks, but the underlying function of the control remains the same. For a complete description of CUA notebooks, refer to the *SAA CUA Guide to User Interface Design* and the *SAA CUA Advanced Interface Design Reference.*

Notebook Creation

You create a notebook by using the WC_NOTEBOOK window class name in the *ClassName* parameter of the WinCreateWindow function. Figure 19-2 on page 19-2 shows the creation of the notebook. The style set in the *ulNotebookStyles* variable (the BKS_* values) specifies that the notebook is to be created with a solid binding and the back pages intersecting at the bottom right corner, major tabs placed on the right edge, tab type square, tab text centered, and status text left-justified. These are the default settings and are given here only to show how notebook styles are set.

```
HWND  hwndNotebook;             /* Notebook window handle            */
ULONG ulNotebookStyles;         /* Notebook window styles            */
HMODULE hmod;                   /* Notebook DLL module handle        */

/*********************************************************************/
/* Set the BKS_style flags to customize the notebook.                */
/*********************************************************************/
ulNotebookStyles =
  BKS_SOLIDBIND      |          /* Use solid binding.                */
  BKS_BACKPAGESBR    |          /* Set back pages to intersect at the */
                                /* bottom right corner.              */
  BKS_MAJORTABRIGHT |           /* Position major tabs on right side. */
  BKS_SQUARETABS     |          /* Make the tabs square.             */
  BKS_TABTEXTCENTER |           /* Center tab text.                  */
  BKS_STATUSTEXTLEFT;           /* Align status line text left.      */

/*********************************************************************/
/* Create the notebook control window.                               */
/*********************************************************************/
hwndNotebook =
  WinCreateWindow(
    hwndParent,                 /* Parent window handle              */
    WC_NOTEBOOK,                /* Notebook window class             */
    NULL,                       /* No window text                    */
    ulNotebookStyles,           /* Notebook window styles            */
    x, y, cx, xy                /* Origin and size                   */
    hwndOwner,                  /* Owner window handle               */
    HWND_TOP,                   /* Sibling window handle             */
    ID_BOOK,                    /* Notebook window ID                */
    NULL,                       /* No control data                   */
    NULL;                       /* No presentation parameters        */

/*********************************************************************/
/* Make the notebook control visible.                                */
/*********************************************************************/
WinShowWindow(
  hwndNotebook,                 /* Notebook window handle            */
  TRUE);                        /* Make the window visible.          */
```

Figure 19-2. Sample Code for Creating a Notebook

Understanding the Default Notebook Style

As specified in the preceding sample code, Figure 19-3 on page 19-3 shows how the default notebook control looks when it is created.

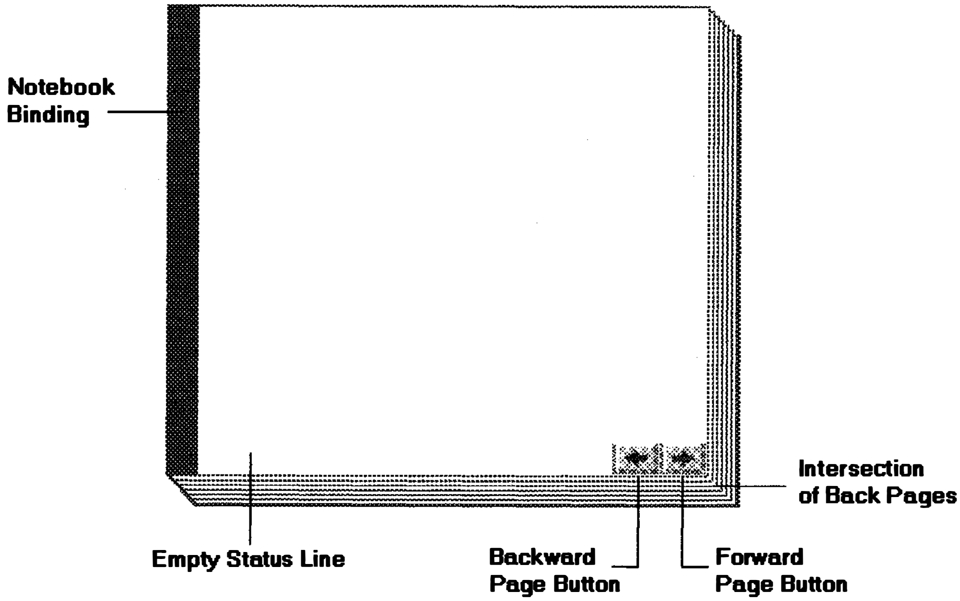


Figure 19-3. Default Notebook Style

The notebook control resembles a real notebook in its general appearance. For example, as Figure 19-3 shows, the notebook has a *binding* that, along with recessed pages on the right and bottom edges, gives the notebook a three-dimensional appearance. The default binding is solid and is placed on the left side. This binding is used if no style bit is specified or if the BKS_SOLIDBIND style bit is specified.

In the bottom right corner of the notebook in Figure 19-3 are the *page buttons*. These buttons are for bringing one page of the notebook into view at a time. They are a standard component that is automatically provided with every notebook. However, the application can change the default width and height of the page buttons by using the BKM_SETDIMENSIONS message.

Selecting the *forward page button* (the arrow pointing to the right) causes the next page to be displayed; while selecting the *backward page button* (the arrow pointing to the left) causes the previous page to be displayed. In Figure 19-3, the page buttons are displayed with unavailable-state emphasis because no pages have been inserted in the notebook yet. Therefore, in this example, selecting either page button would not bring a page into view.

To the left of the page buttons in the default notebook style setting is the *status line*, which enables the application to provide information to the user about the page currently displayed. The notebook does not supply any default text for the status line. The application is responsible for associating a text string with the status line of each page on which a text string is to be displayed. The procedure for associating a text string with a status line is described in "Inserting Notebook Pages" on page 19-8. Text displayed in the status line is left-justified by default. In Figure 19-2 on page 19-2, this setting is specified by the BKS_STATUSTEXTLEFT style bit. See "Notebook Control Styles" on page 19-5 for information about other style bits that can be set for the notebook.

The page buttons always are located in the corner where the recessed edges of the notebook intersect. These recessed edges are called the *back pages*. The default notebook's back pages intersect in the bottom right corner, which means the

recessed pages are on the bottom and right edges. In Figure 19-2 on page 19-2, this setting is specified by the BKS_BACKPAGESBR style bit.

The back pages are important because their intersection determines where the *major tabs* can be placed, which in turn determines the placement of the binding and the *minor tabs*. Major and minor tabs are used to organize related pages into sections; minor tabs define subsections within major tab sections. The content of each section has a common theme, which is represented to the user by a tabbed divider, similar to a tabbed page in a notebook.

In the figure, the BKS_MAJORTABRIGHT style bit specifies that major tabs, if used, are to be placed on the right side of the notebook. This is the default major tab placement when the back pages intersect at the bottom right corner of the notebook. The binding is located on the left, because it is always located on the opposite side of the notebook from the major tabs.

The default notebook shown in Figure 19-3 on page 19-3 has no major tabs, even though the BKS_MAJORTABRIGHT style bit was specified, because major tab attributes, if desired, can be specified only at the time a page is inserted in the notebook. This is done by specifying the BKA_MAJOR attribute in the BKM_INSERTPAGE message.

Similarly, minor tabs are specified using the BKA_MINOR attribute. Minor tabs always are placed perpendicular to the major tabs, based on the intersection of the back pages and the major tab placement. Only one major or minor tab attribute can be specified for each notebook page. Minor tabs are displayed only if the associated major tab page is selected or if the notebook has no major tab pages. Figure 19-4 is an example of a notebook for which both major and minor tab attributes were specified.

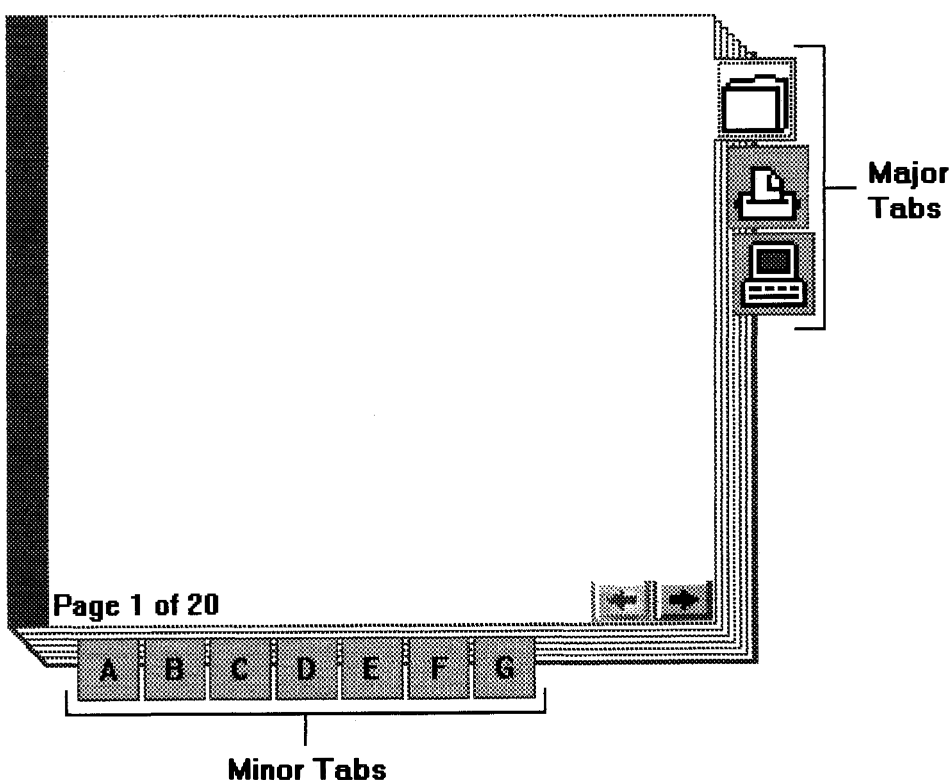


Figure 19-4. Default Style and Placement of Major and Minor Tabs

The default shape of the tabs used on notebook divider pages is square. In Figure 19-2 on page 19-2, this setting is specified by the BKS_SQUARETABS style bit. As with the page buttons, the application can change the default width and height of the major and minor tabs by using the BKM_SETDIMENSIONS message.

A notebook tab can contain either text or a bit map. Text is placed on a tab by using the BKM_TABTEXTCENTER style bit. A bit map is placed on a tab by using the BKM_SETTABBITMAP message. A bit map cannot be positioned on a tab because the bit map stretches to fill the rectangular area of the tab; therefore, no style bit is used.

The following paragraphs provide details about changing notebook style settings, along with additional information about the effect of the back pages intersection on notebook style.

Notebook Control Styles

The notebook control provides style bits so that your application can specify or change the default style settings described in "Understanding the Default Notebook Style" on page 19-2. One style bit from each of the following groups can be specified:

- Type of binding

BKS_SOLIDBIND	Solid (default).
BKS_SPIRALBIND	Spiral.

- Intersection of back pages

BKS_BACKPAGESBR	Bottom right corner (default).
BKS_BACKPAGESBL	Bottom left corner.
BKS_BACKPAGESTR	Top right corner.
BKS_BACKPAGESTL	Top left corner.

- Location of major tabs

BKS_MAJORTABRIGHT	Right edge (default).
BKS_MAJORTABLEFT	Left edge.
BKS_MAJORTABTOP	Top edge.
BKS_MAJORTABBOTTOM	Bottom edge.

- Shape of tabs

BKS_SQUARETABS	Square (default).
BKS_ROUNDEDTABS	Rounded.
BKS_POLYGONTABS	Polygonal.

- Alignment of text associated with tabs

BKS_TABTEXTCENTER	Centered (default).
BKS_TABTEXTLEFT	Left-justified.
BKS_TABTEXTRIGHT	Right-justified.

- Alignment of status line text.

BKS_STATUSTEXTLEFT	Left-justified (default).
BKS_STATUSTEXTRIGHT	Right-justified.
BKS_STATUSTEXTCENTER	Centered.

If you specify more than one style bit, you must use an OR operator (|) to combine them. See the *OS/2 2.0 Programming Reference* for definitions of the notebook style bits.

Two styles are provided for the notebook binding: solid and spiral. The notebook is displayed with a solid binding by default, but the application can specify BKS_SPIRALBIND to display a spiral binding.

The most important of the style bits are those that determine the corner at which the back pages intersect and those that indicate the side where the major tabs are to be placed. For example, if the application specifies the back pages intersection at the bottom right corner (BKS_BACKPAGESBR, the default), the major tabs can be placed on either the bottom edge (BKS_MAJORTABBOTTOM) or the right edge (BKS_MAJORTABRIGHT) of the notebook. In this situation, if the application specifies that major tabs are to be placed on the left or top edges of the notebook, the notebook control places them on the *right* edge anyway,—the default placement for back pages intersecting at the bottom right corner.

The placement of the minor tabs and binding depends entirely on the placement of the back pages and major tabs respectively. The binding *always* is located on the side of the notebook opposite the side where the major tabs are. The minor tabs *always* are located on the recessed page side that has no major tabs. Table 19-1 describes the available notebook window style settings.

Table 19-1. Notebook Window Style Settings

Back Pages	**Major Tabs**	**Minor Tabs**	**Binding**
Bottom right (default)	Bottom	Right	Top
Bottom right (default)	Right (default)	Bottom	Left
Bottom left	Bottom (default)	Left	Top
Bottom left	Left	Bottom	Right
Top right	Top (default)	Right	Bottom
Top right	Right	Top	Left
Top left	Top	Left	Bottom
Top left	Left (default)	Top	Right

The shape of the tabs can be square, rounded, or polygonal. The tab text can be drawn left-justified, right justified, or centered. Once set, these styles apply to the major and minor tabs for all the pages in the notebook. Text is associated with a tab page by using the BKM_SETTABTEXT message. Notebook tab text is centered by default or by specifying the BKS_TABTEXTCENTER style when creating the notebook window.

The application can associate status line text with each inserted notebook page. The status text is drawn left-justified by default, but also can be drawn centered or right-justified. The same status text justification applies to all pages in the notebook. The location of the back pages intersection and the major tabs has no effect on the specification of the tab shape and status line position. These style bits can be set for the entire notebook.

Figure 19-5 shows some sample code for setting the notebook style to spiral binding, back pages that intersect at the bottom left corner, major tabs on the bottom edge, rounded tabs, tab text left justified, and status line text centered.

```
/**********************************************************************/
/* Query for the existing notebook window style settings             */
/**********************************************************************/
ulNotebookStyles = WinQueryWindowULong(
  hwndNotebook,                /* Notebook window handle             */
  QWL_STYLE );                 /* Set notebook style                 */

/**********************************************************************/
/* Reset notebook window style flags, leaving window flags unchanged. */
/**********************************************************************/
ulNotebookStyles &= 0xFFFF0000;

/**********************************************************************/
/* Set up the new notebook window style flags                        */
/**********************************************************************/
ulNotebookStyles |=
  BKS_SPIRALBIND      |        /* Use spiral binding.                */
  BKS_BACKPAGESBL     |        /* Set back pages to intersect at the */
                               /* bottom left corner.                */
  BKS_MAJORTABBOTTOM  |        /* Position major tabs on bottom edge. */
  BKS_ROUNDEDTABS     |        /* Make tabs rounded.                 */
  BKS_TABTEXTLEFT     |        /* Make tab text left justified.      */
  BKS_STATUSTEXTCENTER;        /* Center status text.                */


/**********************************************************************/
/* Set the new notebook style.                                       */
/**********************************************************************/
WinSetWindowULong(
  hwndNotebook,                /* Notebook window handle             */
  QWL_STYLE,                   /* Window style                       */
  ulNotebookStyles);           /* Set notebook style.                */
/**********************************************************************/
/* Invalidate to force a repaint.                                    */
/**********************************************************************/
WinInvalidateRectl(
  hwndNotebook,                /* Notebook window handle             */
  NULL,                        /* Invalidate entire window,          */
  TRUE);                       /* including children.                */
```

Figure 19-5. Sample Code for Changing the Notebook Style

Figure 19-6 shows how the notebook appears when these style bits are set. Compare this figure to the notebook shown in Figure 19-4 on page 19-4. Both of these figures assume that pages have been inserted in the notebook with major and minor tab attributes.

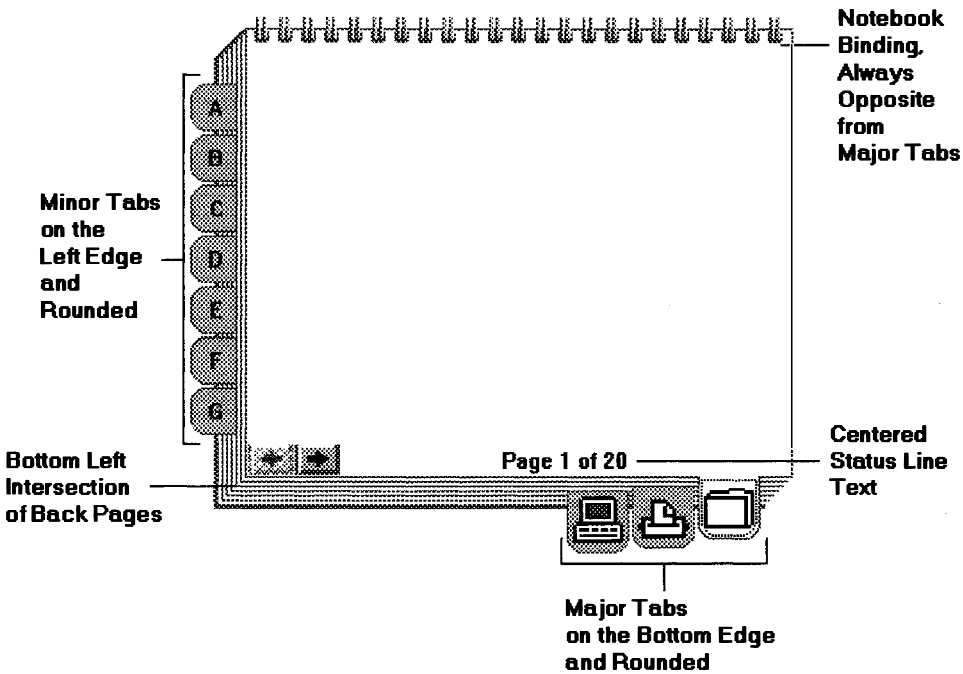


Figure 19-6. Notebook with Style Settings Changed

Working with Notebook Pages and Windows

The following sections tell you how to insert information in, create and associate windows for, and remove information from a notebook.

Inserting Notebook Pages

After a notebook is created, pages can be inserted in the notebook by using the BKM_INSERTPAGE message. BKM_INSERTPAGE provides several attributes that can affect the inserted pages. When inserting pages into either a new notebook or an existing one, carefully consider how the user will expect those pages to be organized.

The two attributes that have the most impact on how notebook pages are organized are BKA_MAJOR and BKA_MINOR, which specify major and minor tabs respectively. Major tab pages define the beginning of major sections in the notebook, while minor tab pages define the beginning of subsections within a major section. Major sections should begin with a page that has a BKA_MAJOR attribute. Within major sections, information can be organized into minor sections, each of which should begin with a page that has a BKA_MINOR attribute.

For an existing notebook, the underlying hierarchy, if one exists, must be observed when inserting new pages, to provide efficient organization and navigation of the information in the notebook.

For example, if the notebook has minor sections but no major sections, you could confuse the user if you inserted a page with a major tab attribute between related minor sections or at the end of the notebook.

If you insert pages without specifying tab attributes, those pages become part of the section in which they are inserted. For example, if page 7 of your notebook has a minor tab, and you insert a new page 8 without specifying a tab attribute, page 8 becomes part of the section that begins with the minor tab on page 7.

Since tab pages are not mandatory, the application can create a notebook that contains no major or minor tab pages. That style would be similar to that of a composition notebook.

Another group of attributes that can affect the organization of pages being inserted into a notebook consists of BKA_LAST, BKA_FIRST, BKA_NEXT, and BKA_PREV. These attributes cause pages to be inserted at the end, at the beginning, after a specified page, and before a specified page of a notebook, respectively.

Each page has an optional status line that can be used to display information for the user. To include this status line, the application must specify the BKA_STATUSTEXTON attribute when inserting the page. If the application inserts the page without specifying this attribute, the status line is not available for that page.

To display text on the status line of the specified page, the application must use the BKM_SETSTATUSLINETEXT message to associate a text string with the page. A separate message must be sent for each page that is to display status line text. If the application does not send a BKM_SETSTATUSLINETEXT message for a page, no text is displayed in the status line of that page. The application can send this message to the notebook at any time to change the status line text. The status line can be cleared by setting the text to NULL.

Figure 19-7 shows how to insert a page in a notebook, where the inserted page has a major tab attribute, the status line is available, and the page is inserted after the last page in the notebook. This sample code also shows how to associate a text string with the status line of the inserted page.

```
HWND  hwndNotebook;              /* Notebook window handle           */
ULONG ulPageId;                  /* Page identifier                  */

/**********************************************************************/
/* Insert a new page into a notebook                                  */
/**********************************************************************/
ulPageId = (ULONG) WinSendMsg(
  hwndNotebook,                  /* Notebook window handle           */
  BKM_INSERTPAGE,                /* Message for inserting a page     */
  (MPARAM)NULL,                  /* NULL for page ID                 */
```

Figure 19-7 (Part 1 of 2). Sample Code for Inserting a Notebook Page

```
  MPFROM2SHORT(
    BKA_MAJOR |                 /* Insert page with a major tab       */
                                /* attribute                          */
    BKA_STATUSTEXTON),          /* Make status line text visible      */
    BKA_LAST));                 /* Insert this page at end of notebook */

/*********************************************************************/
/* Set the status line text.                                         */
/*********************************************************************/
WinSendMsg(
  hwndNotebook,                 /* Notebook window handle             */
  BKM_SETSTATUSLINETEXT,        /* Message for setting status line    */
                                /* text                               */
  (MPARAM)ulPageId,             /* ID of page to receive status line  */
                                /* text                               */
  MPFROMP("Page 1 of 2");       /* Text string to put on status line  */
```

Figure 19-7 (Part 2 of 2). Sample Code for Inserting a Notebook Page

Associating Application Page Windows with Notebook Pages

After a page is inserted into a notebook, you must facilitate the display of information for this page when it is brought to the top of the book. The notebook provides a top page area in which the application can display windows or dialogs for the topmost page. For each inserted page, the application must associate the handle of a window or dialog that is to be invalidated when the page is brought to the top of the book. The application can associate the same handle with different pages if desired.

The application must send a BKM_SETPAGEWINDOWHWND message to the notebook order to associate the application page window or dialog handle with the notebook page being inserted. Once done, the notebook invalidates this window or dialog whenever the notebook page is brought to the top of the book. If no application page window handle is specified for an inserted page, no invalidation can be done by the notebook for that page. However, the application will receive a BKN_PAGESELECTED notification code when a new page is brought to the top of the notebook, at which time the application can invalidate the page.

Associating a Window or Dialog with a Notebook Page

The following sections describe how to associate either a window handle or a dialog handle with an inserted page.

Associating a Window with a Notebook Page

A calendar example is used to show how a page can be implemented as a window. Figure 19-8 shows a calendar that is divided into four years (major tabs). Within each year are months (minor tabs,) grouped into quarters. The top page has a window associated with it. The window paint processing displays the days for the currently selected month and year.

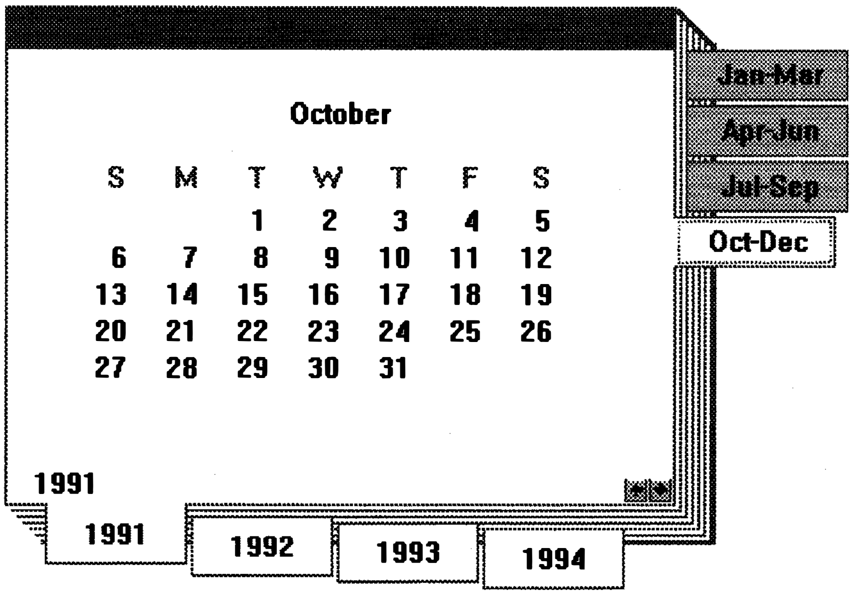


Figure 19-8. Calendar Inserted into an Application Page Window

The sample code in Figure 19-9 shows how the window procedure for the calendar in Figure 19-8 is registered with the application. Also, it shows how the window is created and associated with the notebook page. The example ends by showing the window procedure for the associated window.

```
/**********************************************************************/
/* Registration of window procedure for calendar.                     */
/**********************************************************************/
WinRegisterClass(hab,                 /* Register a page window class */
                 "Calendar Page",     /* Class name                   */
                 PageWndProc,         /* Window procedure             */
                 CS_SIZEREDRAW,       /* Class style                  */
                 0);                  /* No extra bytes reserved      */
```

Figure 19-9 (Part 1 of 3). Sample Code for Associating a Window with a Notebook Page

```
/**********************************************************************/
/* Create the window.                                                 */
/**********************************************************************/
hwndPage = WinCreateWindow(hwndNotebook,       /* Parent              */
                           "Calendar Page",    /* Class               */
                           NULL,               /* Title text          */
                           0L,                 /* Style               */
                           0, 0, 0, 0,         /* Origin and size     */
                           hwndNotebook,       /* Owner               */
                           HWND_TOP,           /* Z-order             */
                           ID_WIN_CALENDAR_PAGE,   /* ID              */
                           NULL,               /* Control data        */
                           NULL);              /* Pres params         */

/**********************************************************************/
/* Associate window with the inserted notebook page.                  */
/**********************************************************************/
WinSendMsg(hwndBook,
           BKM_SETPAGEWINDOWHWND,
           MPFROMLONG(ulPageId),
           MPFROMHWND(hwndPage));

/**********************************************************************/
/* Window procedure.                                                  */
/**********************************************************************/
MRESULT EXPENTRY PageWndProc(HWND hwnd,USHORT msg,MPARAM mp1,
                             MPARAM mp2)
{
HPS hps;

switch (msg)
{
  /********************************************************************/
  /* WM_CREATE is sent when the window is created.                    */
  /********************************************************************/
  case  WM_CREATE :
    /******************************************************************/
    /* Place window initialization code here                          */
    /******************************************************************/
    break;
```

Figure 19-9 (Part 2 of 3). Sample Code for Associating a Window with a Notebook Page

```
    case  WM_PAINT :
      /*****************************************************************/
      /* Draw the calendar for the current selected year and month    */
      /*****************************************************************/
      hps = WinBeginPaint(hwnd, NULL, NULL);
      drawMonthCalendar(hps, windowSize, currDate.year, currDate.month);
      WinEndPaint(hps);
      break;

    default:
      return (WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
  }

  return (FALSE);
  }
```

Figure 19-9 (Part 3 of 3). Sample Code for Associating a Window with a Notebook Page

Associating a Dialog with a Notebook Page
To illustrate the notebook implemented as a dialog, a properties notebook is used. In this example, the various objects whose properties can be changed or updated are displayed as major tabs. Included are sections that represent a folder, a printer, and a display. The printer object is currently selected. Within the printer object, the user can choose to "View" or "Update" the printer settings. The topmost page is a printer dialog from which the user can update the printer name, type, and device information.

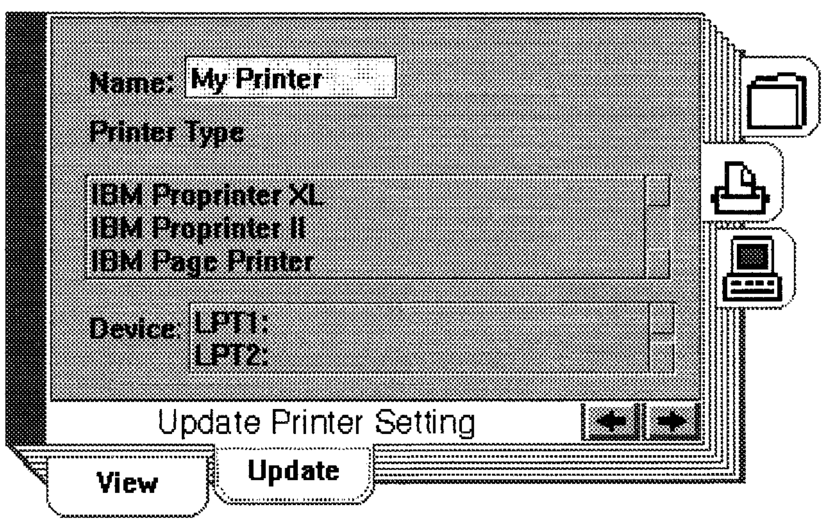


Figure 19-10. Dialog Used As an Application Page Window

The sample code in Figure 19-11 shows how the printer dialog is created and associated with a notebook page. The example ends by showing the dialog procedure for the associated dialog.

```
SEL         sel = NULL;
PDLGTEMPLATE pDlgt;

/*****************************************************************/
/* Create a dialog.                                              */
/*****************************************************************/
DosGetResource(NULL,RT_DIALOG,ID_DLG_PRINTDRV,&sel);
pDlgt = MAKEP(sel,0);
hwndPage = WinCreateDlg(HWND_DESKTOP,     /* Parent window handle */
                        hwndBook,         /* Owner window handle  */
                        fnwpPrint,        /* Dialog procedure     */
                                          /* address              */
                        pDlgt,            /* Dialog structure     */
                                          /* address              */
                        NULL);            /* Application data     */


/*******************************************************************/
/* Associate dialog with the inserted notebook page.               */
/*******************************************************************/
WinSendMsg(hwndBook,
           BKM_SETPAGEWINDOWHWND,
           MPFROMLONG(ulPageId),
           MPFROMHWND(hwndPage));


/*******************************************************************/
/* Dialog procedure                                                */
/*******************************************************************/
MRESULT EXPENTRY fnwpPrint(HWND hwndDlg, USHORT msg, MPARAM mp1,
                           MPARAM mp2)
{
  switch (msg)
  {
    case WM_INITDLG:

      /*************************************************************/
      /* Place dialog initialization code here.                    */
      /*************************************************************/
      break;

    case WM_COMMAND:
      return ((MRESULT) FALSE);
      break;


    default:
      return WinDefDlgProc (hwndDlg,msg,mp1,mp2);
  }
  return WinDefDlgProc (hwndDlg,msg,mp1,mp2);
}
```

Figure 19-11. Sample Code for Associating a Dialog with a Notebook Page

Deleting Notebook Pages

The BKM_DELETEPAGE message is used to delete one or more pages from the notebook. The application can delete one page (BKA_SINGLE attribute), all pages within a major or minor tab section (BKA_TAB attribute), or all of the pages in the notebook (BKA_ALL attribute). The default, if no attributes are specified, is to delete no pages. The following example shows how the BKM_QUERYPAGEID message is used to get the ID of the top page and how the BKM_DELETEPAGE message is then used to delete that page.

```
/**********************************************************************/
/* Set the range of pages to be deleted.                              */
/**********************************************************************/
usDeleteFlag = BKA_SINGLE       /* Set attribute to delete a single   */
                                /* page.                              */

/**********************************************************************/
/* Get the ID of the notebook's top page.                             */
/**********************************************************************/
ulPageId = (ULONG) WinSendMsg(
  hwndNotebook,                 /* Notebook window handle             */
  BKM_QUERYPAGEID,              /* Message to query a page ID         */
  NULL,                         /* NULL for page ID                   */
  (MPARAM)BKA_TOP);             /* Get ID of top page                 */

/**********************************************************************/
/* Delete the notebook's top page.                                    */
/**********************************************************************/
WinSendMsg(
  hwndNotebook,                 /* Notebook window handle             */
  BKM_DELETEPAGE,               /* Message to delete the page         */
  MPFROMLONG(ulPageId),         /* ID of page to be deleted           */
  (MPARAM)usDeleteFlag);        /* Range of pages to be deleted       */
```

Figure 19-12. Sample Code for Deleting a Notebook Page

Graphical User Interface Support

The following describes the support for graphical user interfaces (GUIs) provided by the notebook control. Except where noted, this support conforms to the guidelines in the *SAA CUA Advanced Interface Design Reference.*

The GUI support provided by the notebook control consists of:

- Notebook navigation techniques
- Tailoring notebook colors.

Notebook Navigation Techniques

The notebook control supports the use of a pointing device and the keyboard for displaying notebook pages and tabs, and for moving the selection cursor from the notebook tabs to the application window and the other way around. The following describes this support.

Note: If more than one notebook window is open, displaying a page or tab in one notebook window has no effect on the pages or tabs displayed in any other notebook window.

Pointing Device Support: A user can use a pointing device to display notebook pages or tabs by selecting the notebook components described in the following list. The *SAA CUA Advanced Guide to User Interface Design* defines mouse button 1 (the select button) to be used for selecting these components. This definition also applies to the same button on any other pointing device a user might have.

- **Selecting tabs using a pointing device**

  A tab can be selected to bring a page that has a major or minor tab attribute to the top of the notebook. The *selection cursor*, a dotted outline, is drawn inside the tab's border to indicate the selected tab. In addition, the selected tab is given the same background color as the notebook page area. The color of the other tabs is specified in the BKM_SETNOTEBOOKCOLORS message. This helps the user distinguish the selected tab from the other tabs if different colors are used.

  Since all the tabs are mutually exclusive, only one of them can be selected at a time. Therefore, the only type of selection supported by the notebook control is *single selection*. This selection type conforms to the guidelines in the *SAA CUA Advanced Interface Design Reference*. Refer to that book for detailed information about single selection.

  If the user moves the pointing device to a place in the notebook page window that can accept a cursor, such as an entry field, check box, or radio button, and presses the select button, the selection cursor is removed from the tab it is on and is displayed in the notebook page window. the selection cursor never can be displayed both on a tab and in the notebook page window at the same time.

- **Selecting page buttons using a pointing device**

  A forward or backward *page button* can be selected to display the next or previous page, respectively, one at a time. The arrow pointing to the right is the forward page button, and the arrow pointing to the left is the backward page button. When the selection of a page button brings a page that has a major or minor tab to the top of the notebook, the selection cursor is drawn inside that tab's border. See Figure 19-3 on page 19-3 for an example of page buttons.

- **Selecting tab scroll buttons using a pointing device**

  A user can decrease the size of a notebook window so that some of the available notebook tabs cannot be displayed. When this happens, the notebook control automatically draws *tab scroll buttons* at the corners of the notebook side or sides to notify the user that more tabs are available.

  Tab scroll buttons have another purpose: to give the user the means to scroll into view, one at a time, the tabs that are not displayed. The user does this by selecting a forward or backward tab scroll button, which causes the next tab to scroll into view, but does not change the location of the selection cursor. Once the tab is in view, the user can display that tab's page by selecting the tab.

  A maximum of four tab scroll buttons can be displayed: two for the major tab side and two for the minor tab side. Figure 19-13 is an example of a notebook with two of its tab scroll buttons displayed on the bottom left and bottom right corners of the minor tab side.

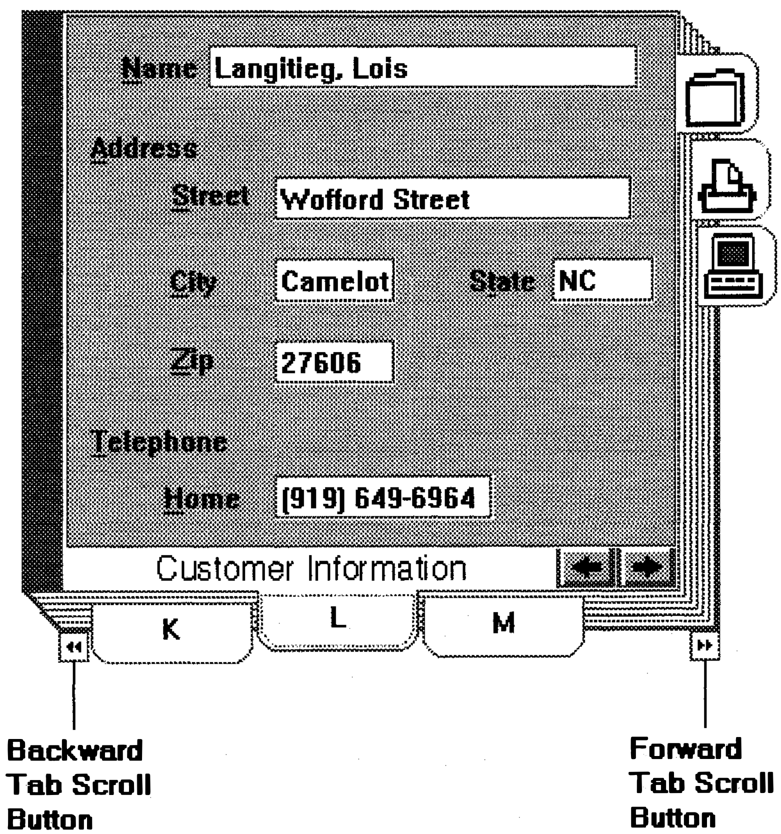


Figure 19-13. Notebook with Tab Scroll Buttons Displayed

In this example, only three minor tabs are displayed because the notebook is not wide enough to display more. Here, the user can display a previous minor tab by selecting the backward tab scroll button or a following minor by selecting the forward tab scroll button.

When the first tab in the notebook is displayed, the backward tab scroll button is deactivated. Unavailable-state emphasis is applied to it to show that no more tabs can be scrolled into view by using the backward tab scroll button. Unavailable-state emphasis is applied to the forward tab scroll button if the last tab in the notebook is displayed.

Keyboard Support
A user can display notebook pages and tabs by using the following keyboard selection techniques.

- **Selecting tabs using mnemonic selection**

  One keyboard method of displaying notebook pages is *mnemonic selection*. Mnemonics are underlined characters in the text of a tab that cause the tab's page to be selected. Coding a tilde (~) before a text character in the BKM_SETTABTEXT message causes that character to be underlined and activates it as a mnemonic-selection character.

  A user performs mnemonic selection by pressing a character key that corresponds to an underlined character. When this happens, the tab that contains the underlined character is selected, and that tab's page is brought to the top of the notebook.

  Note: Mnemonic selection is not case sensitive, so the user can type the underscored letter in either uppercase or lowercase.

- **Selecting tabs using the keyboard**

  Another method of displaying a notebook page is to use the Enter key or the spacebar to select a page with a major or minor tab. The selection cursor, described earlier in this section, indicates that a tab can be selected by using either of these keys. When selected, the tab's associated page is brought to the top of the notebook, and the selected tab is given the same background color as the notebook page area. The other tabs have their color specified in the BKM_SETNOTEBOKKCOLORS message. This helps the user distinguish the selected tab from the other tabs if different colors are used.

- **Moving the selection cursor from tab to tab using the keyboard**

  The selection cursor can be moved from tab to tab by using the Up, Down, Left, and Right Arrow keys. Pressing either the Up or Right Arrow key moves the selection cursor up on a vertical row of tabs or to the right on a horizontal row of tabs. Pressing the Down or Left Arrow keys moves the selection cursor down on a vertical row of tabs or to the left on a horizontal row of tabs. The page associated with the tab to which the selection cursor is moved is not brought to the top of the notebook unless the user selects the tab.

  If the selection cursor is located on a tab that is not in view, pressing one of these keys moves the selection cursor and positions the tab the selection cursor is moved to in the center of the row of tabs.

- **Moving the selection cursor between tab positions and controls**

  Pressing the Tab key moves the selection cursor to the next tab position or control. Pressing the Shift+Tab key combination moves the selection cursor to the previous tab position or control. Pressing Ctrl+Tab moves the selection cursor to the next control.

- **Displaying tabs using the keyboard**

  When the tab scroll buttons are displayed, the Up, Down, Left, and Right Arrow keys can be used to scroll tabs into view. For example, suppose the back pages intersect at the bottom right corner and the selection cursor is on the last visible tab on the right side of the notebook. In this case, pressing either the Down or Right Arrow keys causes the next tab to scroll into view and moves the selection cursor to that tab. The page associated with the tab the selection cursor is moved to is not brought up to the top of the notebook unless the user selects the tab.

- **Turning notebook pages using the keyboard**

  The PgUp and PgDn keys can be used to display the previous or next page, respectively, one page at a time. This is similar to using a pointing device's select button to select the page buttons. The difference is that, unlike the select button, the PgUp and PgDn keys are *typematic*, which means the notebook's pages keep turning while these keys are pressed.

  If the application's primary window has the focus, the PgUp and PgDn keys must be used in combination with the Alt key. The application sends a message to the notebook to turn to the previous or next page. The current top page is used as the page from which to turn.

  If the notebook window has the focus, the PgUp and PgDn keys can be used alone or in combination with the Alt key.

- **Switching the focus between the notebook window and the application's primary window**

  The Alt+Up Arrow key combination switches the focus from the application's primary window to the notebook window. The Alt+Down Arrow key combination does the opposite, switching the focus from the notebook window to the application's primary window.

  If the selection cursor is not in view when the focus switches from the notebook window to the application's primary window, it will not be in view if the focus switches back to the notebook window. For example, the selection cursor may be located on a tab that the user scrolls out of view by selecting a tab scroll button. If the user then presses the Alt+Down key combination, the selection cursor appears in the application's primary window. If the user then presses the Alt+Up Arrow key combination, the selection cursor returns to its last location—the tab that was not in view.

- **Automatic scrolling to the first or last notebook page**

  The Home key can be used to bring the first page of the notebook to the top. Conversely, the End key brings the last page to the top of the notebook.

These selection techniques conform to the descriptions in the *SAA CUA Advanced Guide to User Interface Design*. Refer to that book for a complete description of the keyboard interface model.

Tailoring Notebook Colors

The application can change the color of any part of the notebook. The colors of some parts can be changed by specifying a presentation parameter attribute or attributes in the WinSetPresParam function. Other colors can be changed by specifying a notebook attribute or attributes in the BKM_SETNOTEBOOKCOLORS message. The following sections define which parts of the notebook can have their colors changed by each of these two methods.

Changing Colors Using WinSetPresParam

The WinSetPresParam function is used to change the color of the notebook outline and window background, the selection cursor, and the status line text. The following list shows the mapping between the various notebook parts and their associated presentation parameter attributes.

Notebook outline
: PP_BORDERCOLOR or PP_BORDERCOLORINDEX. This color is set initially to SYSCLR_WINDOWFRAME.

Notebook window background
: PP_BACKGROUNDCOLOR or PP_BACKGROUNDCOLORINDEX. This color is set initially to SYSCLR_FIELDBACKGROUND.

Selection cursor
: PP_HILITEBACKGROUNDCOLOR or PP_HILITEBACKGROUNDCOLORINDEX. This color is set initially to SYSCLR_HILITEBACKGROUND.

Status line text
: PP_FOREGROUNDCOLOR or PP_FOREGROUNDCOLORINDEX. This color is initially set to SYSCLR_WINDOWTEXT.

If a presentation parameter attribute is set, all parts of the notebook that are mapped to this color are changed. Figure 19-14 shows how to change the color of the notebook outline.

```
usColorLen = 4;                 /* Set number of bytes to be passed in */
                                /* usColorIdx for color index table    */
                                /* value.                              */
ulColorIdx = 3;                 /* Set color index table value to be   */
                                /* assigned.                           */


/**********************************************************************/
/* Set the notebook outline color.                                    */
/**********************************************************************/
WinSetPresParam(
  hwndNotebook,                 /* Notebook window handle              */
  PP_BORDERCOLOR,               /* Border color attribute              */
  usColorLen,                   /* Number of bytes in color index      */
                                /* table value                         */
  &ulColorIdx);                 /* Color index table value             */
```

Figure 19-14. Sample Code for Changing the Color of the Notebook Outline

Changing Colors Using BKM_SETNOTEBOOKCOLORS

The BKM_SETNOTEBOOKCOLORS message is used to change the color of the major tab background and text, the minor tab background and text, and the notebook page background. The following list shows the mapping between the various notebook parts and their associated notebook attributes.

Major tab background
: BKA_BACKGROUNDMAJORCOLORINDEX or BKA_BACKGROUNDMAJORCOLOR. This color is set initially to SYSCLR_PAGEBACKGROUND. The currently selected major tab will have the same background color as the page background.

Major tab text
BKA_FOREGROUNDMAJORCOLORINDEX or BKA_FOREGROUNDMAJORCOLOR. This color is set initially to SYSCLR_WINDOWTEXT.

Minor tab background
BKA_BACKGROUNDMINORCOLORINDEX or BKA_BACKGROUNDMINORCOLOR. This color is set initially to SYSCLR_PAGEBACKGROUND. The currently selected minor tab will have the same background color as the page background.

Minor tab text
BKA_FOREGROUNDMINORCOLORINDEX or BKA_FOREGROUNDMINORCOLOR. This color is set initially to SYSCLR_WINDOWTEXT.

Notebook page background
BKA_BACKGROUNDPAGECOLORINDEX or BKA_BACKGROUNDPAGECOLOR. This color is set initially to SYSCLR_PAGEBACKGROUND.

If a notebook attribute is set, all parts of the notebook that are mapped to this color are changed. Figure 19-15 shows how to change the color of the major tab background.

```
ulColorIdx = SYSCLR_WINDOW;                       /* Color index value   */
ulColorRegion = BKA_BACKGROUNDMAJORCOLORINDEX; /* Major tab background*/

WinSendMsg(hwndBook,
           BKM_SETNOTEBOOKCOLORS,
           MPFROMLONG(ulColorIdx),
           MPFROMLONG(ulColorRegion));
```

Figure 19-15. Sample Code for Changing the Color of the Major Tab Background

Enhancing Notebook Control Performance and Effectiveness

This section provides the following information to enable you to fine-tune a notebook control:

- How to dynamically resize and scroll
- How to paint and position tabs

Dynamic Resizing and Scrolling

The notebook control supports *dynamic resizing* by recalculating the size of the notebook's parts when either the user or the application changes the size of any of those parts. A BKN_NEWPAGESIZE notification code is sent from the notebook to the application whenever the notebook's size changes.

The notebook handles the sizing and positioning of each application page window if the BKA_AUTOPAGESIZE attribute is specified for the inserted notebook page. Otherwise, the application must handle this when it receives the BKN_NEWPAGESIZE notification code from the notebook.

If the size of the notebook window is decreased so that the page window is not large enough to display all the information the page contains, the information in the page

window is clipped. If scroll bars are desired to enable the clipped information to be scrolled into view, they must be provided by the application.

Tab scroll buttons are automatically displayed if the size of the notebook is decreased so that all the major or minor tabs cannot be displayed. For example, a notebook has major tabs on the right side, but the height of the notebook does not allow all the tabs to be displayed. In this case, tab scroll buttons are displayed on the upper- and lower-right corners of the notebook. See Figure 19-13 on page 19-17 for an example of tab scroll buttons.

Tab Painting and Positioning

The tab pages provide a method for organizing the information in a notebook so that the user easily can see and navigate to that information. As described in "Notebook Control Styles" on page 19-5, when a page is inserted with a major or minor tab attribute, the notebook displays a tab for that page, based on the orientation of the notebook. The contents of the tab can be painted either by the notebook control or the application.

If the notebook control is to paint the tabs, the application must associate a text string or bit map with the page whose tab is to be drawn. This is done by sending the BKM_SETTABTEXT or BKS_SETTABBITMAP messages to the notebook control for the specified page. If neither of these messages is sent for an inserted page with a major or minor tab attribute, the application must draw the contents of the tab, through *ownerdraw.* The application receives a WM_DRAWITEM message whenever a tab page that has no text or bit map associated with it is to be drawn. The application can either draw the tab contents or return FALSE, in which case the notebook control fills the tab with the tab background color.

Positioning Tabs in Relation to the Top Tab:
There are seven page edges that define the back pages. The page attribute (BKA_MAJOR or BKS_MINOR) and the topmost page determine how the tabs are positioned. In most cases, the tabs must be drawn when their position changes. For example, this can happen when a page with a tab attribute is brought to the top of the notebook.

The new top major or minor tab will appear attached to the top page. The other tabs will appear as described in the following list. This information is provided to help you understand the relationship between the top tab and the other tabs so that you can organize the information you put into a notebook appropriately. The application has no control over tab positioning. See Figure 19-10 on page 19-13 for an example.

- When the top page is a major tab page:
  - Any major tabs prior to the top major tab are aligned on the last page of the notebook.
  - Any major tabs after the top major tab are incrementally cascaded from the topmost edge to the last page.
  - If the top major tab has minor tabs, no major tab is drawn on the page edge that immediately follows the top tab page. Instead, any major tabs that follow the top tab are incrementally cascaded beginning on the second page edge down from the top tab. This is done to account for the minor tabs that are positioned between the top major tab and the major tab that follows it on the perpendicular notebook edge.

    The minor tabs are all positioned on the third page edge from the top, thus giving the appearance of being between the top major tab and the next major tab.
- When the top page is a minor tab page:
  - Any minor tabs prior to the top minor tab are positioned on the third page edge from the top of the notebook.
  - Any minor tabs after the top minor tab are incrementally cascaded up to the third page edge from the top.

Summary

Following are the OS/2 structures, notification codes, notification messages, and window messages used with the notebook control:

Table 19-2. Notebook Control Structures

Structure Name	Description
BOOKTEXT	Contains text strings for notebook status lines and tabs.
DELETENOTIFY	Contains information about the page being deleted from a notebook.
PAGESELECTNOTIFY	Contains information about the page being selected in a notebook.

Table 19-3. Notebook Control Notification Codes

Code Name	Description
BKN_HELP	Indicates that the notebook control has received a WM_HELP message.
BKN_NEWPAGESIZE	Indicates that the dimensions of the notebook page window have changed.
BKN_PAGEDELETED	Indicates that a page has been deleted from the notebook.
BKN_PAGESELECTED	Indicates that a new page has been brought to the top of the notebook.

Table 19-4. Notebook Control Notification Messages	
Message	**Description**
WM_CONTROL	Occurs when a control has a significant event to notify to its owner.
WM_CONTROLPOINTER	Sent to the notebook control's owner window when the pointing device pointer moves over the notebook control window, enabling the owner to set the pointing device pointer.
WM_DRAWITEM	Sent to the owner of the notebook control each time and item is to be drawn.

Table 19-5 (Page 1 of 2). Notebook Control Window Messages	
Message	**Description**
BKM_CALCPAGERECT	Calculates a window rectangle from a notebook rectangle or a notebook rectangle from a window rectangle, depending on the setting of the *fPage* parameter.
BKM_DELETEPAGE	Deletes the specified page or pages from the notebook data list.
BKM_INSERTPAGE	Inserts the specified page into the notebook data list.
BKM_INVALIDATETABS	Repaints all the tabs in the notebook.
BKM_QUERYPAGECOUNT	Queries the number of pages.
BKM_QUERYPAGEDATA	Queries the 4 bytes of application-reserved storage associated with the specified page.
BKM_QUERYPAGEID	Queries the page identifier for the specified page.
BKM_QUERYPAGESTYLE	Queries the style that was set when the specified page was inserted.
BKM_QUERYPAGEWINDOWHWND	Queries the notebook page window handle associated with the specified page.
BKM_QUERYSTATUSLINETEXT	Queries the status line text, text size, or both, for the specified page.
BKM_QUERYTABBITMAP	Queries the bit-map handle associated with the specified page.
BKM_QUERYTABTEXT	Queries the text, text size, or both, for the specified page.
BKM_SETDIMENSIONS	Sets the height and width for the major tabs, minor tabs, or page buttons.
BKM_SETNOTEBOOKCOLORS	Sets the colors for the major tab text and background, minor tab text and background, and notebook page background.
BKM_SETPAGEDATA	Sets the 4 bytes of application-reserved storage associated with the specific page.
BKM_SETPAGEWINDOWHWND	Associates a notebook page window handle with the specified notebook page.

Message	Description
BKM_SETSTATUSLINETEXT	Associates a text string with the status line on the specified page.
BKM_SETTABBITMAP	Associates a bit-map handle with the specified page.
BKM_SETTABTEXT	Associates a text string with the specified page.
BKM_TURNTOPAGE	Brings the specified page to the top of the notebook.
WM_CHAR	Occurs when the user presses a key.
WM_PRESPARAMCHANGED	Occurs when a presentation parameter is set or removed dynamically from a window instance.
WM_SIZE	Occurs when the size of the notebook window changes.

Table 19-5 (Page 2 of 2). Notebook Control Window Messages

Chapter 20. Slider Controls

A slider control (WC_SLIDER window class) is a visual component that enables a user to set, display, or modify a value by moving the slider arm along the slider shaft. This chapter explains how you can use slider controls in your PM applications.

About Slider Controls

Figure 20-1 is an example of a slider used to set a decibel value.

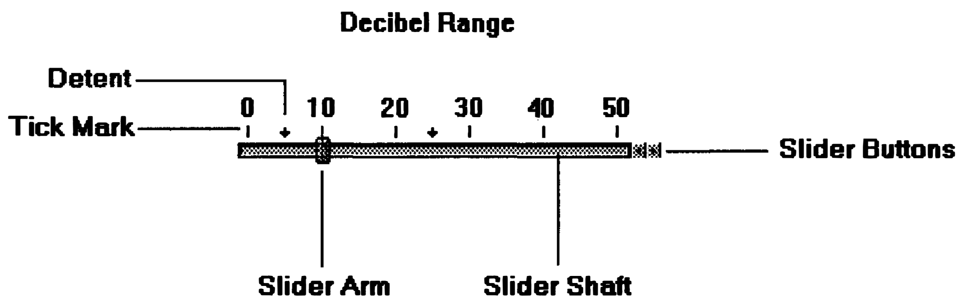


Figure 20-1. Sample Slider

The *slider arm* shows the value currently set by its position on the *slider shaft*. The user selects slider values by changing the location of the slider arm.

A *tick mark* indicates an incremental value in a slider scale. A *detent*, similar to a tick mark, also represents a value on the scale; however, a detent can be placed anywhere along the slider scale, instead of only in specific increments, and can be selected.

Typically, sliders are used to easily set values that have familiar increments, such as feet, inches, degrees, decibels, and so forth. They also can be used for other purposes when immediate feedback is required, such as to blend colors or show a task's percentage of completion. For example, an application might let a user mix and match color shades by moving a slider arm, or a read-only slider could show how much of a task is complete by filling in the slider shaft as the task progresses. These are just a few examples of the many ways in which sliders can be used.

The appearance of and user interaction for a slider is similar to that of a scroll bar. However, these two controls are not interchangeable because each has a unique purpose. A scroll bar scrolls information into view that is outside a window's work area, while the slider is used to set, display, or modify that information, whether it is in or out of the work area.

The slider can be customized to meet varying application requirements, while providing a user interface component that can be used easily to develop products that conform to the Common User Access (CUA) user interface guidelines. The application can specify different scales, sizes, and orientations for its sliders, but the underlying function of the control remains the same. For a complete description of CUA sliders, refer to the *SAA CUA Guide to User Interface Design* and the *SAA CUA Advanced Interface Design Reference.*

Creating a Slider

Before the slider is created, a temporary SLDCDATA data structure is allocated, and variables are specified for the slider control window handle and slider style. The SLDCDATA data structure is allocated so that the scale increments and spacing of the slider can be specified. Refer to the *OS/2 Programming Reference* for more information about the SLDCDATA data structure.

The slider style variable enables the application to specify style bits, SLS_* values, that are used to customize the slider. Refer to the *OS/2 Programming Reference* for the definitions of these values.

You create a slider by using the WC_SLIDER window class name in the *ClassName* parameter of the WinCreateWindow function call. The handle of the slider control window is returned in the slider window variable.

After the slider is created, but before it is made visible, the application can set other slider control characteristics, such as:

- Size and placement of tick marks
- Text above one or more tick marks
- One or more detents
- Initial slider arm position.

The settings in the preceding list are just a few that an application can specify and are the ones shown in the following sample code for creating a slider. Slider control messages are used to specify these settings. A detailed description of the messages is available in the *OS/2 2.0 Programming Reference*.

Figure 20-1 on page 20-1 shows how the slider created by the sample code in Figure 20-2 would appear, except for the *Decibel Range* text string. The code that inserts this static text string is separate from the code used to create a slider window and, therefore, is not included here. The main components of the slider are labeled.

```
SLDCDATA sldcData;                /* SLDCDATA data structure          */
CHAR     szTickText[5];           /* Text strings variable            */
USHORT   idx;                     /* Counter for setting text strings */
HWND     hwndSlider;              /* Slider window handle             */
ULONG    ulSliderStyle;           /* Slider styles                    */

/*********************************************************************/
/* Initialize the parameters in the data structure.                  */
/*********************************************************************/
sldcData.cbSize = sizeof(SLDCDATA); /* Size of SLDCDATA structure    */
sldcData.usScale1Increments = 6;    /* Number of increments          */
sldcData.usScale1Spacing = 0;       /* Use 0 to have slider calculate */
                                    /* spacing                        */
```

Figure 20-2 (Part 1 of 3). Sample Code for Creating a Slider

```
/**********************************************************************/
/* Set the SLS_* style flags to the default values, plus slider      */
/* buttons right.                                                     */
/**********************************************************************/
ulSliderStyle = SLS_HORIZONTAL |    /* Slider is horizontal           */
                SLS_CENTER |        /* Slider shaft centered in       */
                                    /* slider window                  */
                SLS_HOMELEFT |      /* Home position is left edge of  */
                                    /* slider                         */
                SLS_PRIMARYSCALE1 | /* Scale is displayed above       */
                                    /* slider shaft                   */
                SLS_BUTTONSRIGHT;   /* Slider buttons at right end of */
                                    /* slider                         */

/**********************************************************************/
/* Create the slider control window.  The handle of the window is    */
/* returned in hwndSlider.                                            */
/**********************************************************************/
hwndSlider = WinCreateWindow(
               hwndClient,     /* Parent window handle                */
               WC_SLIDER,      /* Slider window class name            */
               (PSZ)NULL,      /* No window text                      */
               ulSliderStyle,  /* Slider styles variable              */
               (SHORT)10,      /* X coordinate                        */
               (SHORT)10,      /* Y coordinate                        */
               (SHORT)150,     /* Window width                        */
               (SHORT)80,      /* Window height                       */
               hwndClient,     /* Owner window handle                 */
               HWND_TOP,       /* Sibling window handle               */
               ID_SLIDER,      /* Slider control window ID            */
               &sldcData,      /* Control data structure              */
               (PVOID)NULL);   /* No presentation parameters          */

/**********************************************************************/
/* Set tick marks at several places on the slider shaft using the    */
/* primary scale.                                                     */
/**********************************************************************/
WinSendMsg(hwndSlider,             /* Slider window handle            */
           SLM_SETTICKSIZE,        /* Message for setting tick mark size*/
           MPFROM2SHORT(
             SMA_SETALLTICKS,      /* Attribute for setting all tick  */
                                   /* marks to the same size          */
             6),                   /* Draw tick marks 6 pixels long   */
           NULL);                  /* Reserved value                  */
```

Figure 20-2 (Part 2 of 3). Sample Code for Creating a Slider

```
/**********************************************************************/
/* Set text above the tick marks.                                     */
/**********************************************************************/
for (idx = 0; idx <= 5; idx++)     /* Count from 0 to 5               */
{
  itoa( 10*idx, szTickText, 10 );  /* Set text at increments of 10    */

  WinSendMsg(hwndSlider,           /* Slider window handle            */
         SLM_SETSCALETEXT,         /* Message for setting text on a   */
                                   /* slider scale                    */
         MPFROMSHORT(idx),         /* Text string counter             */
         MPFROMPSZ(szTickText));   /* Text to put on slider scale     */
}

/**********************************************************************/
/* Set detents between two of the tick marks on the slider shaft.     */
/**********************************************************************/
WinSendMsg(hwndSlider,           /* Slider window handle              */
         SLM_ADDDETENT,          /* Message for adding detents to a   */
                                 /* slider scale                      */
         MPFROMSHORT(5),         /* Put a detent 5 pixels from home   */
         NULL);                  /* Reserved value                    */

WinSendMsg(hwndSlider,           /* Slider window handle              */
         SLM_ADDDETENT,          /* Message for adding detents to a   */
                                 /* slider scale                      */
         MPFROMSHORT(25),        /* Put a detent 25 pixels from home  */
         NULL);                  /* Reserved value                    */

/**********************************************************************/
/* Set the slider arm position to the 1st increment on the scale.     */
/**********************************************************************/
WinSendMsg(hwndSlider,                 /* Slider window handle        */
         SLM_SETSLIDERINFO,            /* Message for setting slider  */
                                       /* attributes                  */
         MPFROM2SHORT(
           SMA_SLIDERARMPOSITION,      /* Modify slider arm position  */
           SMA_INCREMENTVALUE),        /* Use an increment value      */
         MPFROMSHORT(1);               /* Value to use is 1st         */
                                       /* increment                   */

/**********************************************************************/
/* Since all items have been set, make the control visible.           */
/**********************************************************************/
WinShowWindow(hwndSlider,        /* Slider window handle              */
            TRUE);               /* Make the window visible           */
```

Figure 20-2 (Part 3 of 3). Sample Code for Creating a Slider

Retrieving Data for Selected Slider Values

To retrieve data represented by a slider value, specify a variable for the current position of the slider arm. Then, use the SLM_QUERYSLIDERINFO message to retrieve information about the current slider arm position in increment coordinates. The code fragment in Figure 20-3 shows how to retrieve data for a selected slider value.

```
ULONG  ulValue;                  /* Variable in which to store current  */
                                 /* slider arm position                  */

/**********************************************************************/
/* Get the information about the current slider arm position in       */
/* incremental coordinates.                                           */
/**********************************************************************/
ulValue = (ULONG)WinSendMsg(
  hwndSlider,                    /* Slider window handle                 */
  SLM_QUERYSLIDERINFO,           /* Message for querying slider          */
                                 /* attributes                           */
  MPFROM2SHORT(
    SMA_SLIDERARMPOSITION,       /* Get increment at which slider arm   */
    SMA_INCREMENTVALUE),         /* is located                           */
  NULL);                         /* Reserved value                       */
```

Figure 20-3. Retrieving a Slider Value

Graphical User Interface Support for Sliders

This section describes the support the slider control provides for graphical user interfaces (GUIs). Except where noted, this support conforms to the guidelines in the *SAA CUA Advanced Interface Design Reference*.

Since slider values all are mutually exclusive, only one of them can be selected at a time. Therefore, the only type of selection supported by the slider control is *single selection*.

Note: If more than one slider window is open, selecting values in one slider window has no effect on the values selected in any other slider window. A black square is drawn in the center of the slider arm to show which slider control window has the focus.

An initial value is selected when the slider control first is displayed. If the application does not provide the initial selection, using the SLM_SETSLIDERINFO message to position the slider arm, the value at the home position is selected automatically. The *home position* is the end of the slider that contains the lowest value on the scale.

The slider control supports the use of pointing devices and the keyboard for selecting values.

Pointing Device Support

A user can select slider values with a pointing device. On a mouse, the *SAA CUA Guide to User Interface Design* defines button 1 (the select button) as the button for selecting values, and button 2 (the drag button) for dragging the slider arm to a value. These definitions also apply to the same buttons on any other pointing device, such as a joystick.

The select button and drag button can be used in conjunction with the following slider components to select slider values:

- **Slider arm**

  Moving the pointer over the slider arm, then pressing and holding the select or drag buttons while moving the pointer, causes the slider arm to move in the direction the pointer is moving. When the button is released, the value closest to the slider arm position becomes the selected value.

- **Slider shaft**

  Clicking the select button when the pointer is over the slider shaft causes the slider arm to move one increment in the direction of the pointer. Increments are determined by the initial values passed for the primary scale specified (SLS_PRIMARYSCALE1 or SLS_PRIMARYSCALE2) when the slider is created.

  Clicking the drag button when the pointer is over the slider shaft causes the slider arm to move to the pointer's location.

- **Slider buttons**

  Clicking the select button when the pointer is over a slider button causes the slider arm to move one increment in the direction the arrow on the slider button is pointing.

  Slider buttons are optional. If used, two slider buttons are available to the user. The arrows on top of the slider buttons point to opposite ends of the slider. Both slider buttons are positioned at the same end of the slider.

  Slider buttons are enabled by specifying the appropriate SLS_* value when the slider control window is created. For horizontal sliders, you can specify either SLS_BUTTONSLEFT or SLS_BUTTONSRIGHT. For vertical sliders, you can specify either SLS_BUTTONSBOTTOM or SLS_BUTTONSTOP. The default is no slider buttons. If more than one of these style bits is specified, no slider buttons are enabled.

- **Detents**

  A detent is similar to a tick mark on a slider scale because it represents a value on the scale. However, unlike a tick mark, a detent can be placed anywhere along the slider scale instead of in specific increments.

  A detent can be selected by moving the pointer over it and pressing the select button on the pointing device. When this happens, the slider arm moves to the position on the slider shaft indicated by the detent.

Keyboard Support

A user can select a value by using the navigation keys to move the slider arm to the value or by typing a value in an entry field, if one is provided by the application, to change the slider arm position. The following list describes these methods of selecting slider values.

- Values can be selected using the Up, Down, Left, and Right Arrow keys to move the slider arm one increment at a time. The Up and Down Arrow keys are

enabled for vertical sliders, and the Right and Left Arrow keys are enabled for horizontal sliders. If no tick mark exists on the scale in the requested direction, the slider arm does not move.

If an Arrow key is pressed in conjunction with the Shift key, the slider arm moves to the next detent instead of the next tick mark. If no detent exists on the scale in the requested direction, the slider arm does not move.

- The Home and End keys can be used to select the lowest and highest values, respectively, in the scale. If the Ctrl key is pressed in combination with the Home or End keys, the result is the same as pressing only the Home or End keys.
- The application can provide an optional entry field for the slider control. The entry field is a separate control, but it can work in conjunction with the slider control.

  If the application provides an entry field for the slider control window, it must be implemented as follows:

  - The user must be allowed to type a value into the entry field.
  - If the typed value is within the range of the slider scale, the slider arm moves to that value as soon as the value is typed.
  - No other action, such as pressing the Enter key, is required.

These selection techniques conform to the descriptions in the *SAA CUA Guide to User Interface Design*.

Summary

Following are tables that describe the OS/2 functions, structure, notification codes, notification messages, and window messages used with the slider control.

Table 20-1. Slider Control Functions

Function Name	**Description**
WinCreateWindow	Creates a window.
WinSendMsg	Sends a message with identity *Msgid* to *hwnd*.
WinShowWindow	Sets the visibility state of a window.

Table 20-2. Slider Control Structure

Structure Name	**Description**
SLDCDATA	Slider control data structure.

Table 20-3 (Page 1 of 2). Slider Control Notification Codes

Code Name	**Description**
SLN_CHANGE	Sent when the slider arm position has changed.
SLN_KILLFOCUS	Sent when the slider control is losing the focus.
SLN_SETFOCUS	Sent when the slider control is receiving the focus.

Table 20-3 (Page 2 of 2). Slider Control Notification Codes	
Code Name	**Description**
SLN_SLIDERTRACK	Sent when the slider arm is being dragged, but it has not been released.

Table 20-4. Slider Control Notification Messages	
Message	**Description**
WM_CONTROL	Occurs when the slider control has a significant event to notify to its owner.
WM_CONTROLPOINTER	Sent to the owner window of the slider control when the pointing device pointer moves over the slider control window, enabling the owner window to set the pointer.
WM_DRAWITEM	Sent to the owner of the slider control each time an item is to be drawn.

Table 20-5. Slider Control Window Messages	
Message	**Description**
SLM_ADDDETENT	Places a detent along the slider shaft at the position specified on the primary scale.
SLM_QUERYDETENTPOS	Queries for the current position of a detent.
SLM_QUERYSCALETEXT	Queries for the text associated with a tick mark for the primary scale and copies that text into a buffer.
SLM_QUERYSLIDERINFO	Queries the current position or dimensions of a key component of the slider.
SLM_QUERYTICKPOS	Queries for the current position of a tick mark for the primary scale.
SLM_QUERYTICKSIZE	Queries for the size of a tick mark for the primary scale.
SLM_REMOVEDETENT	Removes a previously specified detent.
SLM_SETSCALETEXT	Sets text above a tick mark for the primary scale.
SLM_SETSLIDERINFO	Sets the current position or dimensions of a key component of the slider.
SLM_SETTICKSIZE	Sets the size of a tick mark for the primary scale.
WM_CHAR	Occurs when the user presses a key.
WM_PRESPARAMCHANGED	Sent when a presentation parameter is set or removed dynamically from a window instance.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the window parameters.

Chapter 21. Value Set Controls

A *value set control* (WC_VALUESET window class), like a radio button, is a visual component that enables a user to select one choice from a group of mutually exclusive choices. However, unlike radio buttons, a value set can use graphic images (bit maps or icons), as well as colors, text, and numbers, to represent the items a user can select. This chapter presents the basics about value set controls and tells you how to create and use them in PM applications.

About Value Sets

Even though text is supported, the purpose of a a value set control is to display choices as graphic images for faster selection. The user can *see* the selections instead of having to take time to read descriptions of the choices. Using graphic images in a value set also lets you conserve space on the display screen. For example, if you want to let a user choose from a variety of patterns, you can present those patterns as value set choices, as shown in Figure 21-1, instead of providing a list of radio buttons with a description of each pattern.

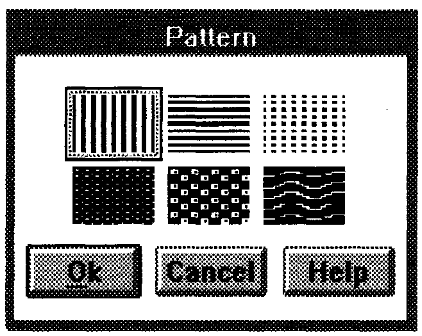


Figure 21-1. Sample Value Set

If long strings of data are to be displayed as choices, radio buttons should be used. However, for small sets of numeric or textual information, you can use either a value set or radio buttons.

The value set is customizable to meet varying application requirements, while providing a user interface component that can be used easily to develop products that conform to the Common User Access (CUA) user interface guidelines. The application can specify different types of items, sizes, and orientations for its value sets, but the underlying function of the control remains the same. For a complete description of CUA value sets, refer to the *SAA CUA Guide to User Interface Design* and the *SAA CUA Advanced Interface Design Reference*.

Creating and Using Value Set Controls

This section provides information that will enable you to create and use a value set control effectively.

Creating a Value Set

You create a value set by using the WC_VALUESET window class name in the *ClassName* parameter of the WinCreateWindow function call.

Before the value set is created, a temporary *VSCDATA* data structure is allocated so that the number of rows and columns of the value set can be specified. Refer to the *OS/2 2.0 Programming Reference* for more information about the *VSCDATA* data structure.

Also, VS_* values are specified in the *ulValueSetStyle* variable so that the value set can be customized. Refer to the *OS/2 2.0 Programming Reference* for descriptions of the value set control styles that can be specified. The sample code in Figure 21-2 shows the creation of a value set.

```
VSCDATA vscData;              /* VSCDATA data structure           */
HWND    hwndValueSet;         /* Value set window handle          */
ULONG   ulValueSetStyle;      /* Value set style variable         */

/**********************************************************************/
/* Initialize the parameters in the data structure.                 */
/**********************************************************************/
vscData.cbSize =              /* Size of value set equals size of  */
  sizeof(VSCDATA);            /* VSCDATA                           */
vscData.usRowCount = 1;       /* 1 row in the value set            */
vscData.usColumnCount = 3;    /* 3 columns in the value set        */

/**********************************************************************/
/* Set the VS_* style flags to customize the value set.             */
/**********************************************************************/
ulValueSetStyle =
  VS_RGB |                    /* Use colors for items.             */
  VS_ITEMBORDER |             /* Put border around each value      */
                              /* set item.                         */
  VS_BORDER;                  /* Put border around the entire      */
                              /* value set.                        */
```

Figure 21-2 (Part 1 of 2). Sample Code for Creating a Value Set

```
/**********************************************************************/
/* Create the value set control window.  The handle of the window is  */
/* returned in hwndValueSet.                                          */
/**********************************************************************/
hwndValueSet = WinCreateWindow(
                  hwndClient,      /* Parent window handle            */
                  WC_VALUESET,     /* Value set class name            */
                  (PSZ)NULL,       /* No window text                  */
                  ulValueSetStyle, /* Value set styles                */
                  (SHORT)10,       /* X coordinate                    */
                  (SHORT)10,       /* Y coordinate                    */
                  (SHORT)300,      /* Window width                    */
                  (SHORT)200,      /* Window height                   */
                  hwndClient,      /* Owner window handle             */
                  HWND_TOP,        /* Z-order position                */
                  ID_VALUESET,     /* Value set window ID             */
                  &vscData,        /* Control data structure          */
                  (PVOID)NULL);    /* No presentation parameters      */


/**********************************************************************/
/* Set the color value for each item in each row and column.          */
/**********************************************************************/
WinSendMsg(hwndValueSet,              /* Value set window handle      */
           VM_SETITEM,                /* Message for setting items    */
           MPFROM2SHORT(1,1),         /* Set item in row 1, column 1  */
           MPFROMLONG(0x00FF0000));   /* to the color red.            */


WinSendMsg(hwndValueSet,              /* Value set window handle      */
           VM_SETITEM,                /* Message for setting items    */
           MPFROM2SHORT(1,2),         /* Set item in row 1, column 2  */
           MPFROMLONG(0x0000FF00));   /* to the color green.          */


WinSendMsg(hwndValueSet,              /* Value set window handle      */
           VM_SETITEM,                /* Message for setting items    */
           MPFROM2SHORT(1,3),         /* Set item in row 1, column 3  */
           MPFROMLONG(0x000000FF));   /* to the color blue.           */


/**********************************************************************/
/* Set the default selection.                                         */
/**********************************************************************/
WinSendMsg(hwndValueSet,              /* Value set window handle      */
           VM_SELECTITEM,             /* Message for selecting items  */
           MPFROM2SHORT(1,2),         /* Item in row 1, column 2      */
           NULL);                     /* Reserved value               */


/**********************************************************************/
/* Since all items have been set in the control, make the control     */
/* visible.                                                           */
/**********************************************************************/
WinShowWindow(hwndValueSet,     /* Value set window handle            */
              TRUE);            /* Make the window visible            */
```

Figure 21-2 (Part 2 of 2). Sample Code for Creating a Value Set

Retrieving Data for Selected Value Set Items

The next step is to be able to retrieve the data represented by a value set item. To do this, variables are specified for combined row and column index values, item attributes, and item information. Then the VM_QUERYSELECTEDITEM, VM_QUERYITEMATTR, and VM_QUERYITEM messages are used to retrieve the index values, attributes, and data. Refer to the descriptions of these messages in the *OS/2 2.0 Programming Reference* for more information. The sample code in Figure 21-3 shows how data for selected value set items is retrieved.

```
ULONG  ulIdx;                   /* Combined row and column index value */
USHORT usItemAttr;              /* Item attributes                     */
ULONG  ulItemData;              /* Item data                           */

/**********************************************************************/
/* Get the row and column index values of the item selected by the    */
/* user.  These values are returned in the ulIdx parameter.           */
/**********************************************************************/
ulIdx = (ULONG)WinSendMsg(
  hwndValueSet,                 /* Value set window handle             */
  VM_QUERYSELECTEDITEM,         /* Message for querying the selected   */
                                /* item                                */
  NULL, NULL);                  /* Reserved values                     */

/**********************************************************************/
/* Determine the type of item that was selected. This message is      */
/* only to determine how to interpret item data when a value set      */
/* contains different types of items.                                 */
/**********************************************************************/
usItemAttr = (USHORT)WinSendMsg(
  hwndValueSet,                 /* Value set window handle             */
  VM_QUERYITEMATTR,             /* Message for querying item attribute */
  MPFROMLONG(ulIdx),            /* Row and column of selected item     */
  NULL);                        /* Reserved value                      */

/**********************************************************************/
/* Get the information about the selected (non-textual) item.  If you */
/* are dealing with text, you need to allocate a buffer for the text  */
/* string.                                                            */
/**********************************************************************/
ulItemData = (ULONG)WinSendMsg(
  hwndValueSet,                 /* Value set window handle             */
  VM_QUERYITEM,                 /* Message for querying an item        */
  MPFROMLONG(ulIdx),            /* Row and column of selected item     */
  NULL);                        /* Set to NULL because the item is not */
                                /* a text item.                        */
```

Figure 21-3. Sample Code for Retrieving Data for Value Set Items

Arranging Value Set Items

The application defines the arrangement of value set items; they can be arranged in one or more rows, columns, or both. Items are placed from left to right in rows and from top to bottom in columns. The application can change the number of rows and columns at any time.

The number of items that can be displayed depends on the number of items that fit into the spaces provided by the defined rows and columns. If the number of items exceeds the number of spaces, the excess items are not displayed.

You can change the composition of a value set by specifying new items. The new items either can be added to the value set or can replace existing items.

Graphical User Interface Support

This section describes the support the value set control provides for graphical user interfaces (GUIs). Except where noted, this support conforms to the guidelines in the *SAA CUA Advanced Interface Design Reference*.

The GUI support provided by the value set control consists of:

- Navigating to and selecting value set items
- Dynamic resizing.

Navigating to and Selecting Value Set Items

Since all value set items are mutually exclusive, only one of them can be selected at a time. Therefore, the only type of selection supported by the value set control is *single selection*. This selection type conforms to the guidelines in the *SAA CUA Advanced Interface Design Reference*. Refer to that book for detailed information about single selection.

Note: If more than one value set window is open, navigating to and selecting items in one value set window has no affect on the items displayed in any other value set window.

An initial choice is selected when the value set control is first displayed. If the application does not provide the initial selection by using the VM_SELECTITEM message, the choice in row 1, column 1 is selected automatically.

The value set control supports the use of a pointing device, such as a mouse, and the keyboard for navigating to and selecting items, except for items that are dimmed on the screen. This dimming of items is called *unavailable-state emphasis* and indicates that the items cannot be selected. However, the *selection cursor*, a dotted outline that usually indicates that an item can be selected, can be moved to unavailable items so that a user can press F1 to determine why they cannot be selected. The following sections describe the pointing device and keyboard support for the value set control.

Pointing Device Support

A user can use a pointing device to select value set items. The *SAA CUA Guide to User Interface Design* defines mouse button 1, the *select* button, to be used for selecting items. This definition also applies to the same button on any other pointing device.

An item can be selected by moving the pointer of the pointing device to the item and clicking the select button. When this happens, a black box is drawn around the item to show that it has been selected. The black box is called *selected-state emphasis*. In addition, the selection cursor is drawn inside the black box.

Keyboard Support

The value set control supports *automatic selection*, which means that an available item is selected when the selection cursor is moved to that item. The item is given selected-state emphasis as soon as the selection cursor is moved to it. No further action, such as pressing the spacebar, is required. The same black box and dotted outline are used, for selected-state emphasis and the selection cursor respectively, as when an item is selected with a pointing device.

A user can navigate to and select an item by using either the navigation keys or mnemonic selection to move the selection cursor to the item, as described in the following list:

- Items can be selected using the Up, Down, Left, and Right Arrow keys to move the selection cursor from one item to another.
- The Home and End keys can be used to select the leftmost and rightmost items, respectively, in the current row. If the Ctrl key is pressed in combination with the Home or End key, the item in the top row and the leftmost column, or the item in the bottom row and the rightmost column, respectively, is selected.

  Note: The preceding description assumes that the current style of the value set window is left-to-right. However, if the VS_RIGHTTOLEFT style bit is set, the directions described for the Home, End, Ctrl+Home, and Ctrl+End keys in the preceding paragraph are reversed.
- The PgUp key can be used to select the item in the top row that is directly above the current position of the selection cursor. The PgDn key can be used to select the item in the bottom row that is directly below the current position of the selection cursor. If the space in the top or bottom row directly above or below the current cursor position is blank, the cursor moves to the blank space.
- Another keyboard method of selecting items is *mnemonic selection*. A user performs mnemonic selection by pressing a character key that corresponds to an underlined character. Coding a tilde (~) before a text character in the item causes that character to be underlined and activates it as a mnemonic selection character. When this happens, the selection cursor is moved to the item that contains the underlined character, and that item is selected.

These selection techniques conform to the descriptions in the *SAA CUA Guide to User Interface Design*. Refer to the *SAA CUA Guide to User Interface Design* for a complete description of the keyboard interface model.

Dynamic Resizing

The value set control supports *dynamic resizing* if the application sends the WM_SIZE message to a value set window. This means that the value set control automatically recalculates the size of the items when either the user or the application changes the size of the value set window.

If the value set window's size is decreased so that the window is not large enough to display all of the items the value set contains, the items are clipped. If scroll bars are desired to allow the clipped information to be scrolled into view, they must be provided by the application.

Summary

The following tables describe the OS/2 structures, functions, notification codes, notification messages, and window messages used with value set controls.

Table 21-1. Value Set Control Structures

Structure Name	Description
VSCDATA	Contains information about the value set control.
VSDRAGINFO	Contains information about direct manipulation actions that occur over the value set control.
VSDRAGINIT	Contains information that is used to initialize a direct manipulation action over the value set control.
VSTEXT	Contains value set text. Used only with the VM_QUERYITEM message.

Table 21-2. Value Set Control Functions

Function Name	Description
WinCreateWindow	Creates a new window.
WinSendMsg	Sends a message to a window.
WinShowWindow	Sets the visibility state of a window

Table 21-3. Value Set Control Notification Codes

Code Name	Description
VN_DRAGLEAVE	Sent when the value set receives a DM_DRAGLEAVE message.
VN_DRAGOVER	Sent when the value set receives a DM_DRAGOVER message.
VN_DROP	Sent when the value set receives a DM_DROPHELP message.
VN_DROPHELP	Sent when the value set receives a DM_DROPHELP message.
VN_ENTER	Sent when the user presses the Enter key while the value set window has the focus, or when the user double-clicks the select button while the pointer is over an item in the value set.
VN_HELP	Sent when the value set receives a WM_HELP message.
VN_INITDRAG	Sent when the drag button is pressed and the pointer is moved while over the value set control.
VN_KILLFOCUS	Sent when the value set loses the focus.
VN_SELECT	Sent when an item in the value set is selected and given selected-state emphasis.
VN_SETFOCUS	Sent when the value set receives the focus.

Table 21-4. Value Set Control Notification Messages	
Message	**Description**
WM_CONTROL	Occurs when the value set control has a significant event to notify to its owner.
WM_CONTROLPOINTER	Sent to the owner window of the value set control when the pointing device pointer moves over the value set control window, enabling the pointer to be set.
WM_DRAWITEM	Sent to the owner of the value set control each time an item is to be drawn.

Table 21-5. Value Set Control Window Messages	
Message	**Description**
VM_QUERYITEM	Queries the contents of the item indicated by the row and column values.
VM_QUERYITEMATTR	Queries the attributes of the item indicated by the row and column values.
VM_QUERYMETRICS	Queries the current size of each value set item or the spacing between items.
VM_QUERYSELECTEDITEM	Queries for the currently selected value set item indicated by the row and column values.
VM_SELECTITEM	Selects the value set item indicated by the row and column values.
VM_SETITEM	Specifies the type of information that will be contained by a value set item.
VM_SETITEMATTR	Sets the attributes of the item indicated by the row and column values.
VM_SETMETRICS	Sets the size of each item in the value set control, the spacing between items, or both.
WM_CHAR	Occurs when the user presses a key.
WM_PRESPARAMCHANGED	Sent when a presentation parameter is set or removed dynamically from a window instance.
WM_QUERYWINDOWPARAMS	Occurs when an application queries the window parameters.
WM_SETWINDOWPARAMS	Occurs when an application sets or changes the window parameters.

Chapter 22. Keyboard Accelerators

A *keyboard accelerator* (*shortcut key* to the user) is a keystroke that generates a command message for an application. This chapter describes how to use keyboard accelerators in your PM applications.

About Keyboard Accelerators

Using a keyboard accelerator has the same effect as choosing a menu item. While menus provide an easy way to learn an application's command set, accelerators provide quick access to those commands.

Without accelerators, a user might generate commands by pressing the Alt key to access the menu bar, using the Arrow keys to select an item, then pressing the Enter key to choose the item. In contrast, accelerators allow the user to generate commands *with a single keystroke.* Figure 22-1 shows examples of accelerators.

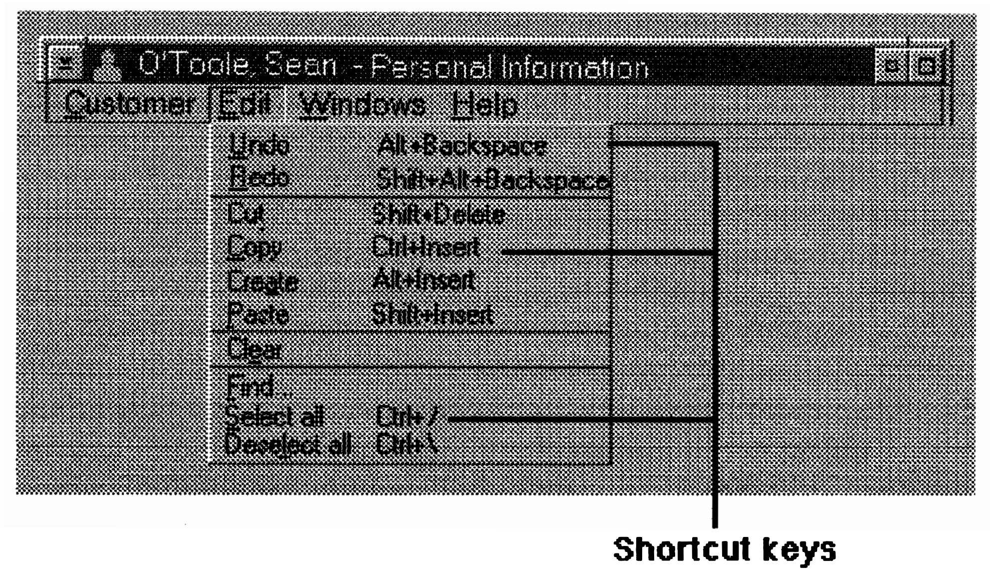


Figure 22-1. Accelerators

Like menu items, accelerators can generate WM_COMMAND, WM_HELP, and WM_SYSCOMMAND messages. Although, normally, accelerators are used to generate existing commands as menu items, they also can send commands that have no menu-item equivalent.

Accelerator Tables

An accelerator table contains an array of accelerators. Accelerator tables exist at two levels within the operating system: a single accelerator table for the system queue and individual accelerator tables for application windows. Accelerators in the system queue apply to all applications – for example, the F1 key always generates a WM_HELP message. Having accelerators for individual application windows ensures that an application can define its own accelerators without interfering with other applications. An accelerator for an application window can override the accelerator in the system queue. An application can modify both its own accelerator table and the system's accelerator table.

The application can set and query the accelerator table for a specific window or for the entire system. For example, an application can query the system accelerator table, copy it, modify the copied data structures; and then, use the modified copy to set the system accelerator table. An application also can modify its window's accelerator table at run time to respond more appropriately to the current environment.

Note: An application that modifies any accelerator table other than its own should maintain the original accelerator table; and, before terminating, restore that table.

Accelerator-Table Resources

You can use accelerators in an application by creating an accelerator-table resource in a resource-definition file. Then, when the application creates a standard frame window, the application can associate that window with the resource.

As specified in a resource-definition file, an accelerator table consists of a list of accelerator items, each defining the keystroke that triggers the accelerator, the command the accelerator generates, and the accelerator's style. The style specifies whether the keystroke is a virtual key, a character, or a scan code, and whether the generated message is WM_COMMAND, WM_SYSCOMMAND, or WM_HELP; WM_COMMAND is the default.

Accelerator-Table Handles

Applications that use accelerator tables refer to them with a 32-bit handle. An application using this handle, by default, can make most API function calls for accelerators without having to account for the internal structures that define the accelerator table. When an application needs to dynamically create or change an accelerator table, it must use the ACCEL and ACCELTABLE data structures.

Accelerator-Table Data Structures

An accelerator table consists of individual accelerator items. Each item in the table is represented by an ACCEL structure that defines the accelerator's style, keystroke, and command identifier. Typically, an application defines these aspects of an accelerator in a resource-definition file, but the ACCEL structure also can be built in memory at run time.

An accelerator table is represented by an ACCELTABLE structure that specifies the number of accelerator items in the table, the code page used for the keystrokes in the accelerator items, and an array of ACCEL structures (one for each item in the table). Applications that use ACCELTABLE structures directly must allocate sufficient memory to hold all the items in the table.

Accelerator-Item Styles

An accelerator item has a style that determines what combination of keys produces the accelerator and what command message is generated by the accelerator. An application can specify the following accelerator-item styles in the **fs** field of the ACCEL structure:

Table 22-1. Accelerator-Item Styles

Style	Description
AF_ALT	Specifies that the user must hold down the Alt key while pressing the accelerator key.
AF_CHAR	Specifies that the keystroke is a character that is translated using the code page for the accelerator table. (This is the default style.)
AF_CONTROL	Specifies that the user must hold down the Ctrl key while pressing the accelerator key.
AF_HELP	Specifies that the accelerator generates a WM_HELP message instead of a WM_COMMAND message.
AF_LONEKEY	Specifies that the user need not press another key while the accelerator key is down. Typically, this style is used with the Alt key to specify that simply pressing and releasing that key triggers the accelerator.
AF_SCANCODE	Specifies that the keystroke is an untranslated scan code from the keyboard.
AF_SHIFT	Specifies that the user must hold down the Shift key when pressing the accelerator key.
AF_SYSCOMMAND	Specifies that the accelerator generates a WM_SYSCOMMAND message instead of a WM_COMMAND message.
VIRTUALKEY	Specifies that the keystroke is a virtual key—for example, the F1 function key.

Using Keyboard Accelerators

This section explains how to perform the following tasks:

- Create an accelerator-table resource.
- Include an accelerator table in a frame window.
- Modify an accelerator table.

Creating an Accelerator-Table Resource

The following code fragment shows a typical accelerator-table resource:

```
ACCELTABLE ID_ACCEL_RESOURCE
BEGIN
    VK_ESC,    IDM_ED_UNDO,  AF_VIRTUALKEY | AF_SHIFT
    VK_DELETE, IDM_ED_CUT,   AF_VIRTUALKEY
    VK_F2,     IDM_ED_COPY,  AF_VIRTUALKEY
    VK_INSERT, IDM_ED_PASTE, AF_VIRTUALKEY
END
```

This accelerator table has four accelerator items. The first one is triggered when the user presses Shift+Esc, which sends a WM_COMMAND message (the default).

An accelerator table in a resource-definition file has an identifier (ID_ACCEL_RESOURCE in the previous example). You can associate an accelerator-table resource with a standard frame window by specifying the table's resource identifier as the *idResources* parameter of the WinCreateStdWindow function.

An application can load an accelerator table resource-definition file automatically when creating a standard frame window, or it can load the resource independently and associate it with a window or with the entire system.

Including an Accelerator Table in a Frame Window

You can add an accelerator table to a frame window either by using the WinSetAccelTable function or by defining an accelerator-table resource (as shown in the previous section) and creating a frame window with the FCF_ACCELTABLE frame style. The second method is shown in the following code fragment:

```
HWND  hwndFrame,hwndClient;
CHAR  szClassName[]="MyClass";
CHAR  szTitle[]="MyWindow";

ULONG flControlStyle = FCF_SIZEBORDER | FCF_ACCELTABLE |
                       FCF_TITLEBAR   | FCF_MENU;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
    WS_VISIBLE,
    &flControlStyle,
    szClassName,
    szTitle,
    0,
    (HMODULE)NULL,
    ID_MENU_RESOURCE,
    &hwndClient);
```

Notice that if you set the *flControlStyle* parameter to the FCF_STANDARD flag, you must define an accelerator-table resource, because FCF_STANDARD includes the FCF_ACCELTABLE flag.

If the window being created also has a menu, the menu resource and accelerator resource must have the same resource identifier; this is because the WinCreateStdWindow function has only one input parameter to specify the resource identifiers for menus, accelerator tables, and icons. If an application creates an accelerator table resource-definition file; then, opens a standard frame window (as shown in the preceding example), the accelerator table is installed automatically in the window's message queue, and keyboard events are translated during the normal processing of events. The application simply responds to WM_COMMAND, WM_SYSCOMMAND, and WM_HELP messages; it does not matter whether these messages come from a menu or an accelerator.

An application also can add an accelerator table to a window by calling the WinSetAccelTable function with an accelerator-table handle and a frame-window handle. The application can call either the WinLoadAccelTable function to retrieve an accelerator table from a resource file or the WinCreateAccelTable function to create an accelerator table from an accelerator-table data structure in memory.

Modifying an Accelerator Table

You can modify an accelerator table, for either your application windows or the system, by doing the following:

1. Retrieve the handle of the accelerator table.
2. Use that handle to copy the accelerator-table data to an application-supplied buffer.

3. Change the data in the buffer.
4. Use the changed data to create a new accelerator table.

Then you can use the new accelerator-table handle to set the accelerator table, as outlined in the following list:

1. Call WinQueryAccelTable to retrieve an accelerator-table handle.
2. Call WinCopyAccelTable with a NULL buffer handle to determine how many bytes are in the table.
3. Allocate sufficient memory for the accelerator-table data.
4. Call WinCopyAccelTable, with a pointer to the allocated memory.
5. Modify the data in the buffer (assuming it has the form of an ACCELTABLE structure).
6. Call WinCreateAccelTable, passing a pointer to the buffer with the modified accelerator-table data.
7. Call WinSetAccelTable with the handle returned by WinCreateAccelTable.

Summary

Following are the OS/2 functions, structures, and messages used with accelerator tables:

Table 22-2. Accelerator-Table Functions

Function name	Description
WinCopyAccelTable	Used to get the accelerator table corresponding to an accelerator-table handle, or to determine the size of the accelerator-table data.
WinCreateAccelTable	Creates an accelerator table from the accelerator definitions in memory.
WinDestroyAccelTable	Destroys an accelerator table.
WinLoadAccelTable	Loads an accelerator table.
WinQueryAccelTable	Queries the window or queue accelerator table.
WinSetAccelTable	Sets the window-accelerator or queue-accelerator table.
WinTranslateAccel	Translates a WM_CHAR message.

Table 22-3. Accelerator-Table Structures

Structure name	Description
ACCEL	Accelerator structure.
ACCELTABLE	Accelerator-table structure.

Table 22-4. Accelerator-Table Messages

Message	Description
WM_QUERYACCELTABLE	Returns the handle to a window's accelerator table.
WM_SETACCELTABLE	Establishes the window accelerator table to be used for translation when the window is active.
WM_TRANSLATEACCEL	Sent to the focus window when a WM_CHAR message occurs.

Chapter 23. Dialog Windows

Dialog windows (also called *dialog boxes*) provide a high-level method for applications to display and gather information. This chapter describes the creation and use of dialog windows and message boxes in your PM applications.

Note: *Dialog windows, dialog boxes*, and *message boxes* all are *secondary windows* to the user.

About Dialog Windows

A dialog window is a temporary window that contains one or more control windows and, typically, is used to display messages to and gather input from the user. An application usually destroys a dialog window immediately after using it.

OS/2 contains many functions and messages that help manage the control windows that make up a dialog window, thus easing the burden of maintaining complex input and output systems.

Modal and Modeless Dialog Windows

Dialog windows can be modal or modeless. A *modal* dialog window requires that the dialog window be dismissed before the user can activate other windows in the same application. Generally, an application uses a modal dialog window to get essential information from the user before proceeding with an operation. A *modeless* dialog window allows the user to activate other windows in the same application without dismissing the dialog window. Both modal and modeless dialog windows allow the user to activate windows in another application before responding to the dialog window.

Modal dialog windows are easier for an application to manage because they are created, perform their task, and are closed, all with a single function call.

Modeless dialog windows require more attention from the application because they exist until explicitly dismissed. Modeless dialog windows provide a more flexible interface, however, by allowing the user to move to other windows in the application before responding to the dialog window.

Dialog Items

A *dialog item* is a child window of the dialog window, which usually is a window of class WC_FRAME. The operating system provides many predefined window classes, called *control windows*, that you can use as dialog items. Figure 23-1 on page 23-2 is an example.

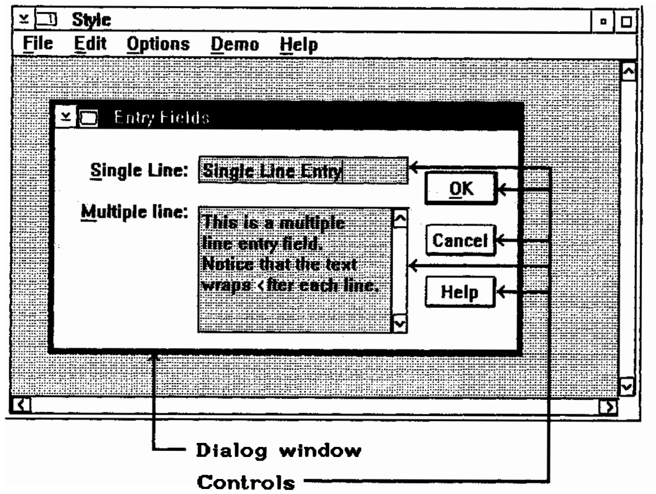


Figure 23-1. Dialog Window with Control Windows

Predefined control windows include static display boxes, text-entry fields, buttons, and list boxes. You also can use customized window classes as dialog items.

Dialog items are windows and, thus, can be manipulated by all window-management functions relating to size, position, and visibility. Dialog items always are owned by the dialog frame window. Most predefined control-window classes send notification messages to their owners when the user interacts with their control windows. The dialog frame window receives these notification messages and passes them to the application through the application-defined dialog procedure.

Dialog-Item Groups

Items within a dialog window can be organized into *dialog-item groups*. When items are arranged in a group, the user can move from one item to another in the same group by using the direction keys. When the user presses a direction key, the focus will not shift to items in other groups within the dialog window.

Arranging items in groups is useful for radio buttons and check boxes. Although some control types also can be displayed this way, entry-field controls cannot; they process direction keys themselves, as do MLE, value-set, container, slider, and notebook controls.

The first item in a dialog-item group has the WS_GROUP window style. All subsequent items in the dialog template are considered part of that group until another item is given the WS_GROUP style, which begins a new group.

The WS_TABSTOP style often is used along with the WS_GROUP style. WS_TABSTOP marks the items that can receive the focus when the user presses the Tab key. Each time the user presses the Tab key, the focus moves to the next item that has the WS_TABSTOP style. Generally, the WS_GROUP and WS_TABSTOP styles are defined together for the first item of each group in the dialog template. This makes it possible for a user to press the Tab key to move among groups of items and to use the direction keys to move among items in a group.

The WS_TABSTOP style should not be used for radio buttons because the system automatically maintains a tab stop on any selected item in a radio-button group; therefore, when the Tab key is pressed in a group of radio buttons, the focus remains on the currently selected item.

The WS_GROUP and WS_TABSTOP styles are also useful for preventing the user from moving to a particular button when using the keyboard. For example, if the dialog window has **OK** and **Cancel** push buttons, they should be in the same group, with the **OK** push button as the first item in the group. The user can press Tab to select the **OK** push button but not the **Cancel** push button. To move to the **Cancel** button using the keyboard, the user first must press the Tab key to move to the **OK** push button, and then press a direction key to move the focus to the **Cancel** push button.

Message Boxes

Message boxes are dialog windows predefined by the system and used as a simple interface for applications, without the necessity of creating dialog-template resources or dialog procedures. An application can call the WinMessageBox function and specify the type of message box and message text. The system displays the message and waits for the user to dismiss the message box by selecting a button in the message box. The system then returns a result code to the application, indicating which button the user selected.

Message boxes are best for short notification messages that require a simple acknowledgment or choice by the user. Applications do not specify a dialog procedure for message boxes so they cannot readily change the action of a message box. However, there is no need to do so, since there are many predefined message-box styles. Figure 23-2 shows a sample message box.

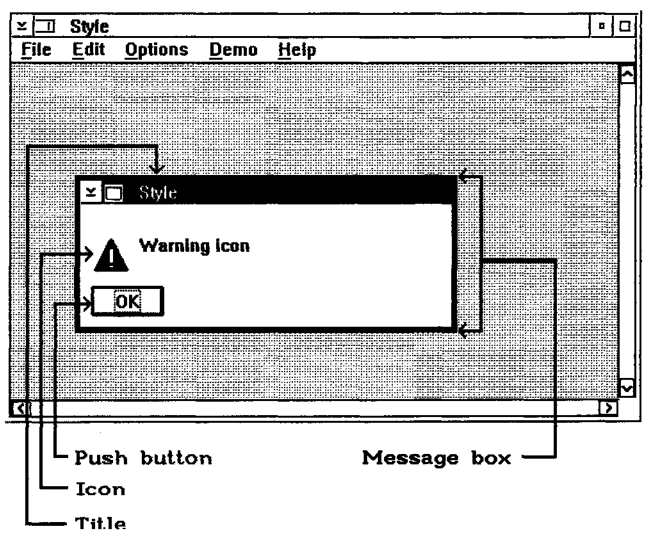


Figure 23-2. Example of a Message Box

Message boxes are always modal—either application-modal or system-modal. *Application-modal* (the default style) means that the user cannot activate another window in the current application before responding to the message box but can switch to another application before responding. *System-modal* means that the user cannot activate another window in any application before responding to the message box. A system-modal message box should be used only to display urgent error messages (running out of memory, for example).

Dialog Data Structures

Each item in a dialog window is described by a DLGTITEM data structure. This structure is rarely accessed directly by an application, since system functions handle most of the manipulation of dialog items. Applications that create dialog items that are not defined as part of a dialog-template resource must create dialog-window-item structures in memory.

A dialog window can have many items, so applications can use another structure, DLGTEMPLATE, to define the items. This structure consists of header information, followed by an array of dialog-window items. Applications that create dialog windows without using dialog resources must create a dialog template in memory, and, then, call the WinCreateDlg function.

Dialog Resources

Most applications define dialog templates in resource files rather than constructing template data structures in memory at run time. The dialog resource file defines the size and style of the dialog-window frame and specifies each dialog item.

The dimensions and position of each dialog item are specified in dialog coordinates, which are based on the size of the system font. A horizontal unit is one-fourth the average width of the characters in the system font; a vertical unit is one-eighth the average height of the characters in the system font. The origin of the dialog template is the lower-left corner of the dialog window. The operating system provides the WinMapDlgPoints function for converting dialog coordinates into window coordinates.

Using Message Boxes and Dialog Windows

The simplest dialog window is the message box. Most message boxes present simple messages and offer the user one, two, or three responses (represented by buttons). A message box is easy to use and is appropriate when an application requires a clearly defined response to a static message. However, message boxes lack flexibility in size and placement on the screen and are limited in the choices they offer the user. Applications that require more control over size, position, and content should use regular dialog windows instead of message boxes.

Creating a Message Box

There are three parts to a message box: the icon, the message, and buttons. Applications specify the icons and buttons by using message-box style constants. Message text is specified by a null-terminated string.

To create a message box, the application calls the WinMessageBox function, which displays the message box and processes user input until the user selects a button in the message box. The WinMessageBox return value indicates which button the user selected.

The following code fragment illustrates how to create a message box with a default **Yes** button, a **No** button, and a question-mark (?) icon. This example assumes that you have defined a string resource with the MY_MESSAGESTR_ID identifier in the resource file.

```
UCHAR  szMessageString[255];
ULONG  ulResult;

WinLoadString(hab, (HMODULE) NULL, MY_MESSAGESTR_ID,
    sizeof(szMessageString), szMessageString);

ulResult = WinMessageBox(hwndFrame,  /* Parent    */
    hwndFrame,                       /* Owner     */
    szMessageString,                 /* Text      */
    (PSZ) NULL,                      /* caption   */
    MY_MESSAGEWIN,                   /* Window ID */
    MB_YESNO |
    MB_ICONQUESTION |
    MB_DEFBUTTON1);                  /* Style     */

 if (ulResult == MBID_YES) {

     /* Do yes case. */

 } else {

     /* Do no case.  */
 }
```

The WinMessageBox function returns predefined values indicating which button has been selected. These values are listed in the *Presentation Manager Programming Reference*.

Notice that strings for message boxes should be defined as string resources to facilitate program translation for other countries. However, there is danger in using string resources in message boxes that are called in low-memory situations; loading a string resource in such situations could result in severe memory problems and cause an application to fail. One way to prevent this problem is to preload the string resource and make it nondiscardable so it will be available when the message box must be displayed.

Creating a System-Modal Message Box

There are two levels of modality for system-modal message boxes—*soft* modal and *hard* modal. A soft-modal message box does not allow keystrokes or mouse input to reach any other window but does allow other messages, such as deactivation and timer messages, to reach other windows. A hard-modal message box does not allow any messages to reach other windows. A hard-model message box is appropriate for serious system warnings.

To create a hard-modal message box, combine the MB_ICONHAND style with the MB_SYSTEMMODAL style. To create a soft-modal message box, use the MB_SYSTEMMODAL style with any style other than MB_ICONHAND. The MB_SYSTEMMODAL icon always is in memory and is available even in low-memory situations.

Using a Dialog Window

When using a dialog window, an application must load the dialog window, process user input, and destroy the dialog window when the user finishes the task. The process for handling a dialog window varies, depending on whether the dialog window is modal or modeless.

Creating a Dialog Template

The following source-code fragment creates a dialog template. Notice that the WS_GROUP and WS_TABSTOP style designations are given for the first item in each group.

```
DLGTEMPLATE IDD_ABOUT
BEGIN
  DIALOG "", IDD_ABOUT2,
  10, 10, 150, 110, FS_DLGBORDER, 0
  BEGIN
    CONTROL "Attributes:",100,
      10, 30, 100, 70,
      WC_STATIC,
      SS_GROUPBOX | WS_VISIBLE
    CONTROL "Highlighted",101,
      20, 80, 58, 12,
      WC_BUTTON,
      WS_GROUP | WS_TABSTOP | BS_AUTOCHECKBOX | WS_VISIBLE
    CONTROL "Enabled",102,
      20, 60, 58, 12,
      WC_BUTTON,
      BS_AUTOCHECKBOX | WS_VISIBLE
    CONTROL "Checked",103,
      20, 40, 58, 12,
      WC_BUTTON,
      BS_AUTOCHECKBOX | WS_VISIBLE
    CONTROL "Okay", DID_OK,
      10, 10, 50, 14,
      WC_BUTTON,
      WS_GROUP | WS_TABSTOP | BS_PUSHBUTTON | BS_DEFAULT | WS_VISIBLE
    CONTROL "Cancel", DID_CANCEL,
      80, 10, 50, 14,
      WC_BUTTON,
      BS_PUSHBUTTON | WS_VISIBLE
  END
END
```

Creating a Modal Dialog Window

The easiest way to use a modal dialog window is to define a dialog template in the resource file (as in the preceding section), and then, call the WinDlgBox function, specifying the dialog-window resource identifier and a pointer to the dialog procedure. WinDlgBox loads the dialog-window resource, displays the dialog window, and handles all user input until the user dismisses the dialog window. The dialog procedure receives messages when the dialog window is created (WM_INITDLG) and other messages each time the user interacts with a dialog item (enters text in entry fields or selects a button, for example).

You must specify both the parent and owner windows when loading a dialog window using the WinDlgBox function. Generally, the parent window will be HWND_DESKTOP and the owner will be a client window in your application.

Dialog windows typically contain buttons that send WM_COMMAND messages when selected by the user. WM_COMMAND messages passed to the WinDefDlgProc function result in the WinDismissDlg function's being called, with the window identifier of the source button as the return code (from WinDismissDlg). Dialog windows with either **OK** or **Cancel** as their only button can ignore WM_COMMAND messages, allowing them to be passed to WinDefDlgProc. WinDefDlgProc calls

WinDismissDlg to dismiss the dialog window and returns the DID_OK or DID_CANCEL code.

Passing WM_COMMAND messages to WinDefDlgProc means that all button presses in the dialog window dismiss the dialog window. If you want certain buttons to initiate operations without closing the dialog window, or if you want to perform some processing without closing the dialog window, handle the WM_COMMAND messages in the dialog procedure.

If you handle WM_COMMAND messages in the dialog procedure, you must call WinDismissDlg to dismiss the dialog window. Your dialog procedure passes the DID_OK code to WinDismissDlg if the user selects the **OK** button or the DID_CANCEL code if the user selects the **Cancel** button.

When you call WinDismissDlg or pass the WM_COMMAND message to WinDefDlgProc, the dialog window is dismissed, and the WinDlgBox function returns the value passed to WinDismissDlg. This return value identifies the button selected.

An alternative to using WinDlgBox is to call the individual functions that duplicate its functionality, as shown in the following code fragment:

```
HWND  hwndDlg;
ULONG ulResult;

hwndDlg = WinLoadDlg(...);
ulResult = WinProcessDlg(hwndDlg);
WinDestroyWindow(hwndDlg);
```

After calling the WinProcessDlg function, your dialog procedure must call WinDismissDlg to dismiss the dialog window. Although the dialog window is *dismissed* (hidden), it still exists. You must call the WinDestroyWindow function to destroy a dialog window if it was loaded using the WinLoadDlg function. WinDlgBox automatically destroys a dialog window before returning.

If you want to manipulate individual items in a dialog window, or add a menu after loading the dialog window (but before calling WinProcessDlg), it is better to make individual calls rather than call WinDlgBox. Individual calls also are useful for querying individual dialog items—to determine the contents of an entry-field control after a dialog window is closed but before it is destroyed, for example. Destroying a dialog window also destroys any dialog-item control windows that are child windows of the dialog window.

Creating a Modeless Dialog Window

To use a modeless dialog window in an application, create a dialog template in the resource file, just as for a modal dialog window. Modeless dialog windows share the screen equally with other frame windows. It is a good idea to give modeless dialog windows a title bar so they can be moved around the screen. The following Resource Compiler source-code fragment shows a dialog template for a dialog window with a title bar, system menu, and minimize button.

```
DLGTEMPLATE IDD_SAMP
BEGIN
    DIALOG "Modeless Dialog", IDD_SAMP, 80, 92, 126, 130,
        WS_VISIBLE | FS_DLGBORDER,
        FCF_TITLEBAR | FCF_SYSMENU | FCF_MINBUTTON

    BEGIN

    /* Put control-window definitions here. */

    END
END
```

The application loads the dialog resource from the resource file using the WinLoadDlg function, receiving in return a window handle to the dialog window. The application treats the dialog window as if it were an ordinary window. Messages for the dialog window are dispatched through the event loop the application uses for its other windows. In fact, an application can have a modeless dialog window as its only window.

The resource for a modeless dialog window is like the resource used for a modal dialog window. The difference between modal and modeless dialog windows is the way applications handle input to each. For a modal dialog, the WinDlgBox and WinProcessDlg functions handle all user input to the dialog window, preventing access to other windows in the application. For a modeless dialog window, the application does not call these functions, relying instead on a normal message loop to dispatch messages to the dialog procedure.

The primary difference between a modeless dialog window and a standard frame window with child control windows is that, for a modeless dialog window, an application can define child windows for the dialog window in a dialog template, automating the process of creating the window and its child windows. The same effect can be achieved by creating a standard frame window, but then, the child control windows must be created individually.

It is important that an application keep track of all open modeless dialog windows so that it can destroy all open windows before terminating.

Initializing a Dialog Window

Generally, an application defines a dialog template in its resource file and loads the dialog window by calling the WinLoadDlg function or the WinDlgBox function (which calls WinLoadDlg). The dialog window is created as an invisible window unless the window style WS_VISIBLE is specified in the dialog template. A WM_INITDLG message is sent to the dialog procedure before WinLoadDlg returns. As each control defined in the template is created, the dialog procedure might receive various control notifications before the function returns. WinLoadDlg returns a handle to the dialog window immediately after creating a dialog window.

In general, it is a good idea to define a dialog window as invisible, since this allows for optimization. For example, an experienced user might type ahead rapidly, anticipating the processing of a dialog-window command. In such a case, there is no need to display the dialog window, because the user has finished the interaction before the window can be displayed. This is how the WinProcessDlg function works—it does not display a dialog window while there still are WM_CHAR messages in the input queue; it lets these messages to be processed first.

As control windows in a dialog window are created from the template, strings in the template are processed by the WinSubstituteStrings function. Any WM_SUBSTITUTESTRING messages are sent to the dialog procedure before WinLoadDlg returns.

When child windows of a dialog window are created, WinSubstituteStrings is used so child windows can make substitutions in their window text. If any child-window text string contains the percent sign (%) substitution character, the length of the text string is limited to 256 characters after it is returned from the substitution.

Adding a Menu in a Dialog Window

To create a menu bar and menus in a dialog window, an application first must load the dialog window to get a handle to the dialog-frame window. The dialog-frame window can be associated with a menu resource by calling the WinLoadMenu function. This function requires arguments that specify the menu identifier and the handle of the parent window for the menu. Finally, the dialog-frame window must incorporate the menu by sending a WM_UPDATEFRAME message to the dialog window. The following code fragment illustrates these operations:

```
HWND hwndDialog, hwndMenu;

/* Get the dialog resource. */
hwndDialog = WinLoadDlg(...);

/ Get the menu resource and attach it to the dialog window. */
hwndMenu = WinLoadMenu(hwndDialog, ...);

/* Inform the dialog window that it has a new menu.         */
WinSendMsg(hwndDialog, WM_UPDATEFRAME, (MPARAM) NULL, (MPARAM) NULL);
```

Applications can create menus in both modal and modeless dialog windows. The preceding code fragment can be used for either type of dialog window. For a modal dialog window, your application must call the WinProcessDlg function to handle user input until the dialog window is dismissed. For a modeless dialog window, your application must call the WinShowWindow function to display the dialog window, enabling the message loop to direct messages to the dialog window.

Creating a Dialog Procedure

The main difference between a dialog procedure and a window procedure is that a dialog procedure does not receive WM_CREATE messages. Instead, a dialog procedure receives WM_INITDLG messages, which are sent after a dialog window is created but before it is displayed. WM_INITDLG can do the same type of initialization tasks that WM_CREATE handles.

For example, if a dialog window contains a list box, use WM_INITDLG to fill the list box with items. Also use this procedure to enable or disable buttons in a dialog window, depending on your application.

You also can call the WinSetDlgItemText or WinSetDlgItemShort functions during dialog initialization, to set up text items that reflect the current conditions in your application.

Another typical task for the WM_INITDLG message handler is centering a dialog window on the screen or within its owner window. The following code fragment illustrates how to center a dialog window on the screen using WM_INITDLG:

```
RECTL rclScreen,rclDialog;
LONG  sWidth,sHeight,sBLCx,sBLCy;

case WM_INITDLG:
    /* Center the dialog window and get the screen rectangle.   */
    WinQueryWindowRect(HWND_DESKTOP, &rclScreen);

    /* Get the dialog-window rectangle.                          */
    WinQueryWindowRect(hwnd, &rclDialog);

    /* Get the dialog-window width.                              */
    sWidth = (LONG) (rclDialog.xRight - rclDialog.xLeft);

    /* Get the dialog-window height.                             */
    sHeight = (LONG) (rclDialog.yTop - rclDialog.yBottom);

    /* Set the horizontal coordinate of the lower-left corner.  */
    sBLCx = ((LONG) rclScreen.xRight - sWidth) / 2;

    /* Set vertical coordinate of the lower-left corner.        */
    sBLCy = ((LONG) rclScreen.yTop - sHeight) / 2;

    /* Move, size, and show the window.                          */
    WinSetWindowPos(hwnd,
        HWND_TOP,
        sBLCx, sBLCy,
        0, 0,           /* Ignores size arguments                */
        SWP_MOVE);

    return 0;
```

The dialog procedure receives notification messages from each control-window item in a dialog window whenever a user clicks an item or enters text in an entry field. Most dialog procedures wait for the user to select one or more dialog-window buttons to signal being finished with the dialog window. When the dialog procedure receives one of these messages, it calls the WinDismissDlg function, as shown in the following code fragment. The second argument to WinDismissDlg is the value returned by the WinDlgBox or WinProcessDlg functions. Generally, these functions return the identifier of the button that was pressed.

```
MRESULT EXPENTRY SampDialogProc(HWND hwnd,
                                ULONG ulMessage,
                                MPARAM mp1,
                                MPARAM mp2)
{
    switch (ulMessage) {
        case WM_COMMAND:
            switch (SHORT1FROMMP(mp1)) {
                case DID_OK:

                   /*
                    * Final dialog-item queries,
                    * dismiss the dialog.
                    */

                   WinDismissDlg(hwnd, DID_OK);
                   return 0;
            }
            break;
    }
    return (WinDefDlgProc(hwnd, ulMessage, mp1, mp2));
}
```

Other dialog-window items send notification messages specific to the type of control window. Make your dialog procedure respond to notification messages from each dialog item. Pass any messages that a dialog procedure does not handle to the WinDefDlgProc function for default processing. The default dialog procedure is the same as the default frame-window procedure.

The WM_COMMAND message from the **OK** button indicates that the user has selected the **OK** button and is finished with the dialog window. If the dialog window has other controls, such as entry fields or check boxes, have your dialog procedure query the contents or state of each control upon receipt of a message from the **OK** button. Before dismissing a dialog window, have your dialog procedure collect input from each dialog-window control before closing the dialog window.

Manipulating Dialog Items

Dialog items are control windows and, as such, can be manipulated using standard window-management function calls. The window handle is obtained for each dialog item by calling the WinWindowFromID function and passing the window handle for the dialog window and the window identifier for the dialog item as defined in the dialog template. Include the following Resource Compiler source-code fragment in your dialog template:

```
DLGTEMPLATE IDD_ABOUT
BEGIN
    DIALOG "", IDD_ABOUT, 80, 92, 126, 130, FS_DLGBORDER, 0
    BEGIN
        PUSHBUTTON "My Button", ITEMID_MYBUTTON, 37, 107, 56, 12

        /* Other item definitions ... */

    END
END
```

Based on this code fragment, your application will receive the button-item handle by initiating the following call to WinWindowFromID:

```
hwndItem = WinWindowFromID(hwndDialog, ITEMID_MYBUTTON);
```

Applications often change the contents, enabled state, or position of dialog items at run time. For example, in a dialog window that contains a list box of file names and an **Open** button, the **Open** button should be disabled until the user selects a file from the list. To do this, define the button as disabled in the dialog resource so that it is disabled when the dialog window first is displayed. At run time, the dialog procedure receives a notification message from the list box when the user selects a file. At that time, the dialog procedure should call the WinEnableWindow function to enable the **Open** button.

Applications also can change the text in static dialog items and buttons by calling the WinSetWindowText function and using the window handle of a particular dialog item.

Summary

Following are the OS/2 functions, structures, and messages used with dialog windows.

Table 23-1 (Page 1 of 2). Dialog Functions

Function name	Description
WinAlarm	Generates an audible alarm.
WinCreateDlg	Creates a dialog window.
WinDefDlgProc	Invokes the default dialog procedure.
WinDestroyWindow	Destroys a window and its child windows.
WinDismissDlg	Hides the modeless dialog window, or destroys the modal dialog window, and causes the WinProcessDlg or WinDlgBox calls to return.
WinDlgBox	Loads and processes a modal dialog window and returns the result value established by the WinDismissDlg call.
WinEnumDlgItem	Returns the window handle of a dialog item within a dialog window.
WinGetDlgMsg	Obtains a message from the application's queue associated with the specified dialog.
WinLoadDlg	Creates a dialog window from the dialog template **Dlgid** in **Resource**.
WinMapDlgPoints	Maps points from dialog coordinates to window coordinates or from window coordinates to dialog coordinates.
WinMessageBox	Creates, displays, and operates a message box window.
WinProcessDlg	Dispatches messages while a modal dialog window is displayed.

Table 23-1 (Page 2 of 2). Dialog Functions	
Function name	**Description**
WinQueryDlgItemShort	Converts the text of a dialog item into an integer value.
WinQueryDlgItemText	Queries a text string in a dialog item.
WinQueryDlgItemTextLength	Queries the length of the text string in a dialog item.
WinSendDlgItemMsg	Sends a message to the dialog item defined by **Item** in the dialog window specified by **Dlg**.
WinSetDlgItemShort	Converts an integer value into the text of a dialog item.
WinSetDlgItemText	Sets a text string in a dialog item.
WinSubstituteStrings	Performs a substitution process on a text string, replacing specific marker characters with text supplied by the application.

Table 23-2. Dialog Structures	
Structure name	**Description**
DLGTEMPLATE	Dialog-template structure.
DLGITEM	Dialog-item structure.

Table 23-3. Dialog Messages	
Message	**Description**
WM_CHAR	Sent when a user presses a key.
WM_INITDLG	Occurs when a dialog box is being created
WM_QUERYDLGCODE	Sent by the dialog manager to identify the type of control, to determine what kinds of messages the control understands, and to determine whether an input message may be processed by the dialog manager or passed down to the control.
WM_SUBSTITUTESTRING	Sent from the WinSubstituteStrings call.

Chapter 24. Font Dialog Controls

Font dialog controls provide basic functions that give users the ability to display and select from a list of:

- Font family names installed on the system
- Available styles for each font
- Available sizes for each font
- Emphasis styles available for each font.

Users can view their selections, using a sample character string in a preview area, and interact with a modal or modeless font dialog. This chapter explains how font dialog controls can be extended to meet the requirements of PM applications.

About the Font Dialog Control

In the font dialog control, *family face* is defined as the name of the typeface. Figure 24-1 is an example of a font dialog.

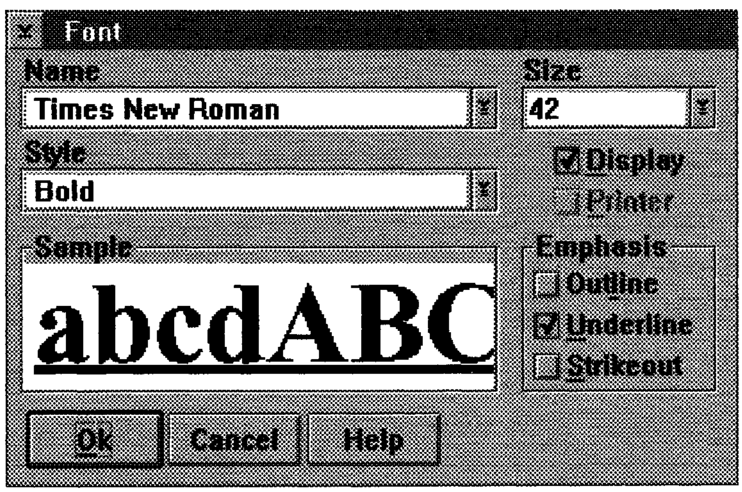


Figure 24-1. Font Dialog

Courier, Times New Roman, and Helvetica are examples of commonly used family faces. Type styles include normal, **bold**, *italic*, and ***bold italic***. *Size* is the point size, or vertical measurement, of the type. Font emphasis styles include outline, underline, and strikeout.

Creating a Font Dialog

To present a font dialog to users, your application must do the following:

1. Allocate storage for a FONTDLG structure and set all fields to NULL.
2. Initialize the fields in the FONTDLG structure.

   The application must:

   a. Set the **cbSize** field to the size of the structure.

   b. Set either the **hpsScreen** or the **hpsPrinter** presentation space field, or both. You must have a valid presentation space from which to query fonts.

c. Pass the pointer to a buffer in which to return the family name selected (**pszFamilyname**) and the size of the buffer (**usFamilyBufLen**). If the application requires a default font, pass the family name of the font in this buffer.

The application can choose to set the following:

a. An application-specific title. Pass the pointer to a null-terminated string in the **pszTitle** field.

b. An application-specific preview string. Pass the pointer to a null-terminated string in the **pszPreview** field.

c. Application-specific available font sizes for outline fonts. Pass the pointer to a null-terminated string containing point sizes, separated by spaces in the **pszPtSizeList** field.

d. A custom dialog procedure to provide application-specific function. Pass the pointer to a window procedure in the **pfnDlgProc** field.

e. Set the appropriate FNTS_* flags in the **fl** field to customize the dialog style. See the description of this field for the FONTDLG structure in the *OS/2 2.0 Programming Reference* for a list of the flags you can specify.

f. Set the FNTF_NOVIEWPRINTERFONTS or FNTF_NOVIEWSCREENFONTS flags to customize the dialog style when working with printer fonts in the **flFlags** field. These filter flags should be initialized only when both the **hpsScreen** and the **hpsPrinter** presentation space fields are non-NULL.

g. Pass the initial position of the dialog in the **x** and **y** fields.

3. Initialize the FONTDLG structure with any values that users should see when they invoke the dialog for the first time. For example, you can:

   a. Pass the characteristics of the default font in the **usWeight**, **usWidth**, **flType**, and **sNominalPointSize** fields.

   b. Pass any display options of the default font in the **flStyle** field.

   c. Pass the color options for displaying the font sample in the **clrFore** and **clrBack** fields.

4. Invoke the font dialog. Call the WinFontDlg function and pass the dialog's parent window handle, owner window handle, and a pointer to the initialized FONTDLG structure.

5. Check the return value from the WinFontDlg function. If it is successful, the selected font can be used by the application. The information returned in the **fAttrs** field of the FONTDLG structure is used.

Graphical User Interface Support for the Font Dialog

Name Field: The **Name** field is a drop-down list that displays a font family name. When the font dialog is invoked, the value displayed in this field is either an application-supplied family name or the default system font.

When users select a family name from the drop-down list, the **Name** field display is refreshed with the selected family name. The preview area is updated to show the sample character string in the selected family face, using the font style, size, and emphasis currently in effect.

Style Field: The **Style** field is a drop-down list that displays a font style. When the font dialog is invoked, the value displayed in this field is either an application-specified font style or the system default.

When users select a font style from the drop-down list, the **Style** field display is refreshed with the selected style name. The preview area is updated to show the sample character string in the selected font style, using the family name, size, and emphasis currently in effect.

Size Field: The **Size** field is a drop-down combination box that displays available font sizes. Users can display and select from a list of available sizes for a font, or they can type a font size directly into the entry field.

When users select a font size from the drop-down list, the **Size** field display is refreshed with the selected size. The preview area is updated to show the character string in the selected font size, using the family name, font style, and emphasis currently in effect.

The font sizes included in the drop-down list are dependent on the character definition of the font. For image or raster fonts, all available sizes are listed. For outline fonts, the default sizes are 8, 10, 12, 14, 18, and 24 points. If required, the application can specify the available sizes for outline fonts.

When users type a font size in the entry field, the preview area is updated immediately. The **Size** field will accept a fixed point number, such as 24.25, with up to four places saved after the decimal.

Emphasis Group Box: The **Emphasis group box** is a multiple-selection field that contains a list of emphasis styles (*Outline, Underline, Strikeout*) available for each font.

When users select an emphasis style, the preview area is updated immediately. The Outline selection is not available for image fonts.

Preview Area: The **Preview** area enables users to view their font family, style, size and emphasis selections as they make them. It contains a sample character string that is defined by the application. The default character string is abcdABCD. The Preview area displays font sizes as large as 48 points. As the size of the font increases, the sample displayed is clipped by the borders of the area.

Filter Check Box: The **Filter** check box enables users to limit the font family name drop-down list to select from fonts that are displayable only, printable only, or a merged list. The initial setting of the **Filter** check box is specified by the application.

Standard Push button and Default Action: The dialog can be dismissed with either the **OK** or **Cancel** push buttons.

Customizing the Font Dialog

You can create a font dialog by customizing the font dialog control, using the minimum set of standard controls and adding any controls of your own design. Specify a standard control by including a control of the same class, ID, and style as in the font dialog. The minimum set of controls required for the font dialog are: DID_NAME, DID_STYLE, DID_DISPLAY_FILTER, DID_PRINTER_FILTER, DID_SIZE, DID_SAMPLE, DID_OUTLINE, DID_UNDERSCORE, DID_STRIKEOUT, DID_OK_BUTTON, DID_CANCEL_BUTTON.

Even if your dialog does not use all of the required controls, you must include them. You can make the unused controls invisible so that your application users are not confused.

Summary

The following tables describe the OS/2 structures, messages, functions, and controls in the standard font dialog:

Table 24-1. Font Dialog Structures

Structure Name	Description
FONTDLG	Font-dialog structure.
STYLECHANGE	Style-change structure returned by the FNTM_STYLECHANGED message.

Table 24-2. Font Dialog Messages

Message Name	Description
FNTM_FACENAMECHANGED	Notifies the subclassing application whenever the font family name is changed by the user.
FNTM_FILTERLIST	Sent whenever the font dialog is preparing to add a font family name, font style type, or point size entry to the combination box fields that contain these parameters.
FNTM_POINTSIZECHANGED	Notifies subclassing applications when the point size of the font is changed by the user.
FNTM_STYLECHANGED	Notifies subclassing applications when the user changes any of the attributes in the STYLECHANGE structure.
FNTM_UPDATEPREVIEW	Notifies subclassing applications before the preview window is updated.

Table 24-3. Font Dialog Functions

Function Name	Description
WinDefFontDlgProc	The default dialog procedure for the font dialog.
WinFontDlg	Allows the user to select a font.

Table 24-4 (Page 1 of 3). Standard Font Dialog Controls

Control Name	ID	Class/Style	Remarks
DID_OK_BUTTON	DID_OK	WC_BUTTON, BS_PUSHBUTTON| BS_DEFAULT| WS_GROUP| WS_TABSTOP| WS_VISIBLE	Button control. Used as an **OK** push button.
DID_CANCEL_BUTTON	DID_CANCEL	WC_BUTTON, BS_PUSHBUTTON| WS_VISIBLE	Button control. Used as a **Cancel** push button.

Table 24-4 (Page 2 of 3). Standard Font Dialog Controls			
Control Name	**ID**	**Class/Style**	**Remarks**
DID_FONT_DIALOG	300	DIALOG, FS_NOBYTEALIGN| FS_DLGBORDER| FS_BORDER| WS_CLIPSIBLINGS| WS_SAVEBITS, FCF_SYSMENU| FCF_TITLEBAR	Dialog control ID.
DID_NAME	301	WC_COMBOBOX, CBS_DROPDOWNLIST| WS_TABSTOP| WS_VISIBLE	Combination box control. Used to display and select font family names.
DID_STYLE	302	WC_COMBOBOX, CBS_DROPDOWNLIST| WS_TABSTOP| WS_VISIBLE	Combination box control. Used to display and select font style names.
DID_DISPLAY_FILTER	303	WC_BUTTON, BS_AUTOCHECKBOX| WS_TABSTOP| WS_GROUP| WS_VISIBLE	Button control. Used to filter the **Font Name** field.
DID_PRINTER_FILTER	304	WC_BUTTON, BS_AUTOCHECKBOX| WS_TABSTOP| WS_VISIBLE	Button control. Used to filter the **Font Name** field.
DID_SIZE	305	WC_COMBOBOX CBS_DROPDOWN| WS_TABSTOP| WS_VISIBLE	Combination box control. Used to display, select, and enter the type size of the selected font.
DID_SAMPLE	306	WC_STATIC, SS_TEXT| DT_CENTER| DT_VCENTER| WS_GROUP| WS_VISIBLE	Static text control. Used to display the preview string in the selected font.
DID_OUTLINE	307	WC_BUTTON, BS_AUTOCHECKBOX| WS_TABSTOP| WS_VISIBLE	Check box control. Used to select the outline emphasis of the selected font.
DID_UNDERSCORE	308	WC_BUTTON, BS_AUTOCHECKBOX| WS_VISIBLE	Check box control. Used to select the underscore emphasis of the selected font.
DID_STRIKEOUT	309	WC_BUTTON, BS_AUTOCHECKBOX| WS_VISIBLE	Check box control. Used to select strikeout emphasis of the selected font.
DID_HELP_BUTTON	310	WC_BUTTON, BS_PUSHBUTTON| BS_HELP| BS_NOPOINTERFOCUS| WS_VISIBLE	Button control. Used to request help from the application.

Table 24-4 (Page 3 of 3). Standard Font Dialog Controls			
Control Name	**ID**	**Class/Style**	**Remarks**
DID_APPLY_BUTTON	311	WC_BUTTON, BS_PUSHBUTTON| WS_VISIBLE	Button control *provided by the application.* Used as an **Apply** push button in modeless applications.
DID_RESET_BUTTON	312	WC_BUTTON, BS_PUSHBUTTON| WS_VISIBLE	Button control *provided by the application.* Used as a **Reset** push button.
DID_NAME_PREFIX	313	WC_STATIC, SS_TEXT| DT_LEFT| DT_TOP| WS_GROUP| WS_VISIBLE	Static text control. Label for the font **Family Name** field.
DID_STYLE_PREFIX	314	WC_STATIC, SS_TEXT| DT_LEFT| DT_TOP| WS_GROUP| WS_VISIBLE	Static text control. Label for the font **Style Name** field.
DID_SIZE_PREFIX	315	WC_STATIC, SS_TEXT| DT_LEFT| DT_TOP| WS_GROUP| WS_VISIBLE	Static text control. Label for the font **Type Size** field.
DID_SAMPLE_GROUPBOX	316	WC_STATIC, SS_GROUPBOX| WS_GROUP| WS_VISIBLE	Group box around a sample field.
DID_EMPHASIS_GROUPBOX	317	WC_STATIC, SS_GROUPBOX| WS_GROUP| WS_VISIBLE	Group box around the emphasis check boxes.

Chapter 25. File Dialog Controls

File dialog controls provide basic functions that enable users to do the following:

- Display and select from a list of drives, directories, and files.
- Enter a file name directly.
- Filter the file names before they are displayed.
- Display active network connections.
- Specify .TYPE EA extended attributes.
- Interact with a single-selection or multiple-selection file dialog.
- Interact with a modal or modeless file dialog.

These basic functions can be extended to meet the requirements of PM applications.

About File Dialogs

The file dialog control enables you to implement *Open* or *SaveAs* dialogs. The following figures illustrate these two dialogs.

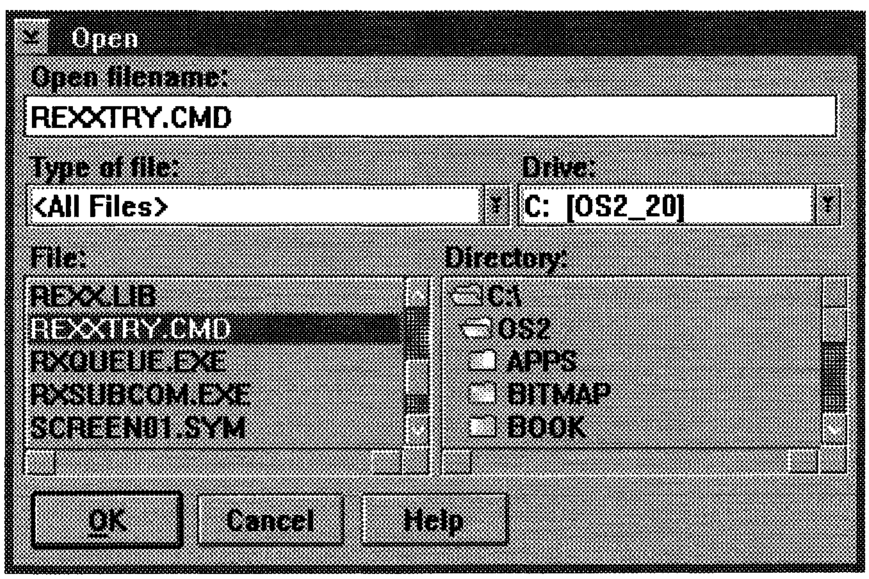


Figure 25-1. Open Dialog

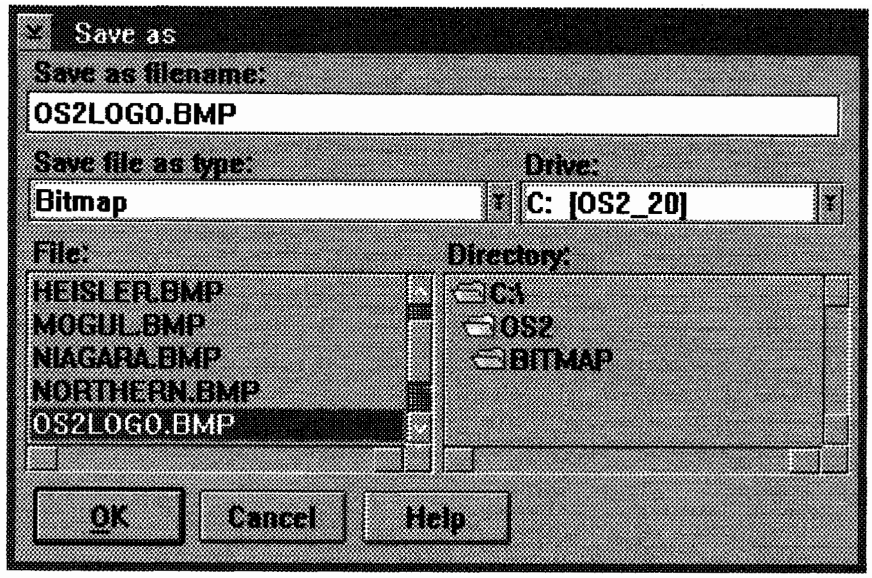


Figure 25-2. SaveAs Dialog

Creating a File Dialog

To present a file dialog to users, your application must do the following:

1. Allocate storage for a FILEDLG structure and set all fields to NULL.
2. Initialize the fields in the FILEDLG structure.

   The application must do the following:

   a. Set the **cbSize** field to the size of the structure.

   b. Set the **fl** field to indicate the type of dialog. You must set the FDS_OPEN_DIALOG or FDS_SAVEAS_DIALOG flags.

   The application can set the following:

   a. An application-specific title. Pass the pointer to a null-terminated string in the **pszTitle** field.

   b. Application-specific text for the **OK** push button. Pass the pointer to a null-terminated string in the **pszOKButton** field.

   c. Specify a custom dialog procedure to provide application-specific function. Pass the pointer to a window procedure in the **pfnDlgProc** field.

   d. Set other FDS_* flags in the **fl** field to customize the dialog style. See the description of this field for the FILEDLG structure in the *OS/2 2.0 Programming Reference* for a list of the flags you can specify.

   e. Pass the initial position of the dialog in the **x** and **y** fields.

3. Initialize the FILEDLG structure with any values that users should see when they invoke the dialog for the first time. For example, you can:

   a. Pass the name of the first drive from which file information will be displayed in the **pszIDrive** field.

   b. If you want to limit user selections, pass a list of drives from which the user can choose in the **papszIDriveList** field. Otherwise, the system defaults to showing all available drives.

c. Pass the name of an extended-attribute filter to be used to filter file information in the **pszIType** field.

d. Pass a list of extended attributes in the **papszITypeList** field. By selecting from this list, users can filter file information.

e. Pass the name of the initial file to be used by the dialog in the **szFullFile** field. This can be a file name or a string filter, such as *.•.dat, to filter the initial file information. This field can be fully qualified to select the initial drive and directory.

4. Invoke the file dialog. Call the WinFileDlg function and pass the dialog's owner window handle and a pointer to the initialized FILEDLG structure.
5. Check the return value from the WinFileDlg function. If TRUE is returned, the application can create the file dialog (either Open or SaveAs) by using the file name or file names returned from the dialog.

Creating an Open Dialog

When the Open dialog is invoked, the fields in the dialog box are updated with the fields passed in the FILEDLG structure. The values passed in the **szFullFile** field of the structure are displayed in the **File Name** field, the Directory list box, and the **Drive** field. The value passed in the **pszIType** field is displayed in the **Type** field.

Creating a SaveAs Dialog

The SaveAs dialog is identical to the Open dialog with these exceptions:

- By default, the file names in the file list box are grayed and cannot be selected, although the list box can be scrolled.
- When the user clicks on the **OK** push button or presses the Enter key, the file name in the **File Name** field is passed to the application, and the application saves, rather than opens, the file.
- The titles of the file name, filter, and dialog are SaveAs rather than Open.

The File Dialog User Interface

File Name Field

The **File Name** field is a single-line entry (SLE) field used to display the name of a file that was selected from the file list box or entered directly by the user. As the user types, the file or files matching the user entry are scrolled into view in the file list box. The first file name that most closely matches the file name typed by the user is placed at the top of the list box. When the user types a character that causes a mismatch, the file at the top of the list is displayed.

When the user presses the Enter key, the dialog returns the selected file name to the application. The application then initiates the default action of opening the file. When a file name is not valid, such as when the file does not exist, the application displays an error message.

The **File Name** field displays the currently selected file name or the current string filter. When a filter is specified in the **szFullFile** field of the FILEDLG structure, the string filter is displayed without the path information. The string filter remains in the field until a file is selected or the user types over the data in the field.

When a file name is not specified, the **File Name** field is blank.

File List Box

The File list box is a single- or multiple-selection list box that is scrollable both horizontally and vertically. It contains all the files that meet the filter criteria, sorted by name.

When the file dialog is a single-selection dialog, the selected file name is placed in the **File Name** field. When the file dialog is a multiple-selection dialog, the topmost selected file name is placed in the **File Name** field. When the user double clicks on a file name, the dialog exits and returns the selected file or files to the application for opening.

Directory List Box

The Directory list box is a single-selection list box that is scrollable both horizontally and vertically.

The Directory list box displays the path in the **szFullFile** field of the FILEDLG structure as a list of each parent subdirectory. Any subdirectories of the selected directory also are displayed. Each directory level is indented to show the path, and the current working directory level is indicated by an arrow. The top entry is always the root directory, with the drive specification preceding it. When the **szFullFile** field is null, the current path of the current drive is displayed. The user selects a new subdirectory by double-clicking on the subdirectory name. This action updates the Directory list box.

Drive Field

The **Drive** field contains a drop-down list of the logical drives. This field cannot be edited by the user.

The **Drive** field displays the value passed in the **papszlDriveList** field of the FILEDLG structure. If the application does not specify a drive list, all drives currently available on the system are displayed. When the drop-down list is displayed, the current drive is highlighted. When the user selects a drive, the display is refreshed. When either the user-specified drive or the default drive has a volume label, the volume label is displayed also.

Users can access networked files by associating logical disks with remote servers, or they can enter the name and ID of the server in the **File Name** field. When the server name entered is not found in the Drive drop-down list, it is added to the list and displayed in the **Drive** field.

Type Field

The **Type** field contains a drop-down list of extended-attribute filters.

The **Type** field displays the value passed in the **pszlType** field of the FILEDLG structure. The current setting is highlighted when the drop-down list is displayed.

When a type filter is not specified by the application, `<All Files>` is displayed and no extended-attribute type filtering is used with the initial display.

All files affected by the string filter and the extended-attribute type filter criteria are displayed, based on how the filters are to be used. The default is that all file names meeting the intersection of the two filters are shown. When users change the value in the **Type** field, the File list box is updated to display a list of files that meet the new type filter criteria. Files that meet both the string filter and extended-attribute type filter are displayed.

Standard Button and Default Action

The **OK** push button initiates the default action.

When a subdirectory is selected, the **File Name** field is empty. When the user clicks on the **OK** push button or presses the Enter key, the subdirectory is opened and the displayed values in the File list box and the Directory list box are refreshed.

When a file name is selected, selection of subdirectories is cancelled and the **File Name** field is updated with the name of the selected file. When the user clicks on the **OK** push button or presses the Enter key, the file displayed in the **File Name** field is returned to the application for opening.

Customizing the File Dialog

You can customize the File Dialog control by using the minimum set of standard controls and adding any of your own design. Specify a standard control by including the control name, ID, and style in the dialog.

Summary

The following tables describe the OS/2 structure, messages, functions, and minimum set of standard controls in the file dialog control:

Table 25-1. File Dialog Structure

Structure Name	Description
FILEDLG	File-dialog structure.

Table 25-2. File Dialog Messages

Message	Description
FDM_ERROR	Sent before the file dialog displays a message notifying the user of an error.
FDM_FILTER	Sent before a file that meets the current filter criteria is added to the File list box.
FDM_VALIDATE	Sent when the user selects a file and presses the Enter key or clicks on the **OK** push button, or when the user double-clicks on a file name in the File list box.

Table 25-3. File Dialog Functions

Function Name	Description
WinDefFileDlgProc	The default dialog procedure for the file dialog.
WinFileDlg	Creates and displays the file dialog and returns the user's selection or selections.
WinFreeFileDlgList	Frees the storage allocated by the file dialog when the FDS_MULTIPLESEL dialog flag is set.

Table 25-4 (Page 1 of 2). File Dialog Minimum Set of Standard Controls			
Control Name	**ID**	**Class/Style**	**Remarks**
DID_OK_PB	DID_OK	WC_BUTTON, BS_PUSHBUTTON| BS_DEFAULT| WS_GROUP| WS_TABSTOP| WS_VISIBLE	Button control. Used as an **OK** push button.
DID_CANCEL_PB	DID_CANCEL	WC_BUTTON, BS_PUSHBUTTON| WS_VISIBLE	Button control. Used as a **Cancel** push button.
DID_FILE_DIALOG	256	DIALOG, FS_NOBYTEALIGN| FS_DLGBORDER| WS_CLIPSIBLINGS| WS_SAVEBITS, FCF_SYSMENU| FCF_TITLEBAR| FCF_DLGBORDER	Dialog control ID.
DID_FILENAME_TXT	257	WC_STATIC, SS_TEXT| DT_LEFT| DT_TOP| WS_GROUP| WS_VISIBLE	Static text control. Label for the **File Name** field.
DID_FILENAME_ED	258	WC_ENTRYFIELD, ES_AUTOSCROLLBAR| ES_LEFT| ES_MARGIN| WS_TABSTOP| WS_VISIBLE	Static entry field. Fully-qualified file name entry field for parsing or selecting.
DID_DRIVE_TXT	259	WC_STATIC, SS_TEXT| DT_LEFT| DT_TOP| WS_GROUP| WS_VISIBLE	Static text control. Label for the **Drive** field.
DID_DRIVE_CB	260	WC_COMBOBOX, CBS_DROPDOWNLIST| WS_TABSTOP| WS_VISIBLE	Combination box control. Used to display and select drive names.
DID_FILTER_TXT	261	WC_STATIC, SS_TEXT| DT_LEFT| DT_TOP| WS_GROUP| WS_VISIBLE	Static text control. Label for the **Type** field.
DID_FILTER_CB	262	WC_COMBOBOX, CBS_DROPDOWNLIST| WS_TABSTOP| WS_VISIBLE	Combination box control. Used to display and select extended-attribute type filters.

Table 25-4 (Page 2 of 2). File Dialog Minimum Set of Standard Controls			
Control Name	**ID**	**Class/Style**	**Remarks**
DID_DIRECTORY_TXT	263	WC_STATIC, SS_TEXT| DT_LEFT| DT_TOP| WS_GROUP| WS_VISIBLE	Static text control. Label for the Directory list box.
DID_DIRECTORY_LB	264	WC_LISTBOX, LS_OWNERDRAW| LS_HORZSCROLL| WS_TABSTOP| WS_VISIBLE	List box control. Used to display and select the directories on the system.
DID_FILES_TXT	265	WC_STATIC, SS_TEXT| DT_LEFT| DT_TOP| WS_GROUP| WS_VISIBLE	Static text control. Label for the Files list box.
DID_FILES_LB	266	WC_LISTBOX, LS_HORZSCROLL| WS_TABSTOP| WS_VISIBLE	List box control. Used to display and select the files in a directory.
DID_HELP_PB	267	WC_BUTTON, BS_PUSHBUTTON| BS_HELP| BS_NOPOINTERFOCUS| WS_VISIBLE	Button control. Used to request help from the application.
DID_APPLY_PB	268	WC_BUTTON, BS_PUSHBUTTON| WS_VISIBLE	Button control. Used to apply selection for a modeless dialog.

Chapter 26. Mouse Pointers and Icons

A *mouse pointer* is a special bit map the operating system uses to show a user the current location of the mouse on the screen. When the user moves the mouse, the mouse pointer moves on the screen. Mouse pointers also are used to draw icons on the screen, such as graphics in message boxes and icons that represent minimized windows on the desktop. This chapter describes how to create and use mouse pointers and icons in your PM applications.

About Mouse Pointers and Icons

Mouse pointers and icons are made up of bit maps that the operating system uses to paint images of the pointers or icons on the screen. A *monochrome bit map* is a series of bytes. Each bit corresponds to a single pel in the image. (The bit map representing the display typically has four bits for each pel.)

A mouse pointer or icon bit map always is twice as tall as it is wide. The top half of the bit map is an *AND* mask, in which the bits are combined, using the AND operator, with the screen bits where the pointer is being drawn. The lower half of the bit map is an *XOR* mask, in which the bits are combined, using the XOR operator, with the destination screen bits.

The combination of the AND and XOR masks results in four possible colors in the bit map. The pels of an icon or pointer can be black, white, transparent (the screen color beneath the pel), or inverted (inverting the screen color beneath the pel). Figure 26-1 shows the relationship of the bit values in the AND and XOR masks:

AND mask	0	0	1	1
XOR mask	0	1	0	1
Result	Black	White	Transparent	Inverted

Figure 26-1. Bit Values in the AND and XOR Masks

Mouse-Pointer Hot Spot

Each mouse pointer has its own *hot spot*, which is the point that represents the exact location of the mouse pointer. This location is defined as an *x* and *y* offset from the lower-left corner of the mouse-pointer bit map.

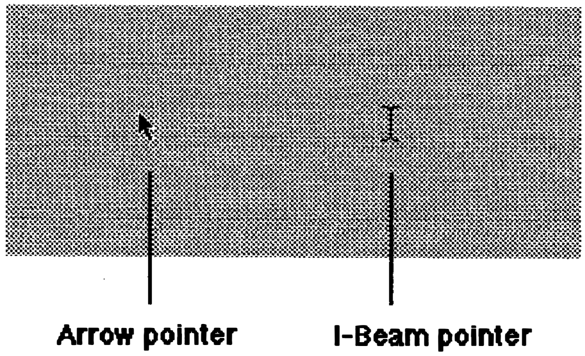


Figure 26-2. Mouse Pointers

For the arrow-shaped pointer, the hot spot is at the tip of the arrow. For the I-beam pointer, the hot spot is at the middle of the vertical line.

Predefined Mouse Pointers

Before an application can use a mouse pointer, it first must receive a handle to the pointer. Most applications load mouse pointers from the system or from their own resource file. The operating system maintains many predefined mouse pointers that an application can use by calling the WinQuerySysPointer function. System mouse pointers include all the standard mouse-pointer shapes and message-box icons. The following predefined mouse pointers are available:

Table 26-1 (Page 1 of 2). Predefined Mouse Pointers

Mouse Pointer	Description
SPTR_APPICON	Square icon; used to represent a minimized application window.
SPTR_ARROW	Arrow that points to the upper-left corner of the screen.
SPTR_ICONERROR	Icon containing an exclamation point; used in a warning message box.
SPTR_ICONINFORMATION	Octagon-shaped icon containing the image of a human hand; used in a warning message box.
SPTR_ICONQUESTION	Icon containing a question mark; used in a query message box.
SPTR_ICONWARNING	Icon containing an asterisk; used in a warning message box.
SPTR_MOVE	Four-headed arrow; used when dragging an object or window around the screen.
SPTR_SIZE	Small box within a box; used when resizing a window by dragging.
SPTR_SIZENS	Two-headed arrow that points up and down (north and south); used when sizing a window.
SPTR_SIZENESW	Two-headed diagonal arrow that points to the upper-right (northeast) and lower-left (southwest) window borders; used when sizing a window.

Table 26-1 (Page 2 of 2). Predefined Mouse Pointers	
Mouse Pointer	**Description**
SPTR_SIZENWSE	Two-headed diagonal arrow that points to the upper-left (northwest) and lower-right (southeast) window borders; used when sizing a window.
SPTR_SIZEWE	Two-headed arrow that points left and right (west to east); used when sizing a window.
SPTR_TEXT	Text-insertion and selection pointer, often called the *I-beam pointer*.
SPTR_WAIT	Hourglass; used to indicate that a time-consuming operation is in progress.

The operating system contains a second set of predefined mouse pointers that are used as icons in PM applications. An application can use one of these icons by supplying one of the following constants in the WinQuerySysPointer function. Before terminating, however, the application must use the WinDestroyPointer function to explicitly destroy the mouse pointer.

Table 26-2. Presentation Manager Mouse Pointers	
Icon	**Description**
SPTR_FILE	Represents a file (in the shape of a single sheet of paper).
SPTR_FOLDER	Represents a file folder.
SPTR_ILLEGAL	Circular icon containing a slash; represents an illegal operation.
SPTR_MULTFILE	Represents multiple files.
SPTR_PROGRAM	Represents an executable file.

Applications can use mouse-pointer resources to draw icons. The WinDrawPointer function draws a specified mouse pointer in a specified presentation space. Many of the predefined system mouse pointers are standard icons displayed in message boxes.

In addition to using the predefined pointer shapes, an application also can use pointers that have been defined in a resource file. Once the pointer or icon has been created (by Icon Editor or a similar application), the application includes it in the resource file, using the POINTER statement, a resource identifier, and a file name for the Icon Editor data. After including the mouse-pointer resource, the application can use the pointer or icon by calling the WinLoadPointer function, specifying the resource identifier and module handle. Typically, the resource is in the executable file of the application, so the application simply can specify NULL for the module handle to indicate the current application resource file.

An application can create mouse pointers at run time by constructing a bit map for the pointer and calling the WinCreatePointer function. This function, if successful, returns the new pointer handle, which the application then can use to set or draw the pointer. The bit map must be twice as tall as it is wide, with the first half defining the AND mask and the second half defining the XOR mask. The application also must specify the hot spot when creating the mouse pointer.

System Bit Maps

In addition to using the mouse pointers and icons defined by the system, applications can use standard system bit maps by calling the WinGetSysBitmap function. This function returns a bit map handle that is passed to the WinDrawBitmap function or to one of the Gpi bit-map functions. The system uses standard bit maps to draw portions of control windows, such as the system menu, minimize/maximize box, and scroll-bar arrows. The following standard system bit maps are available:

Table 26-3 (Page 1 of 2). Standard System Bit Maps

Bit Map	Description
SBMP_BTNCORNERS	Specifies the bit map for push button corners.
SBMP_CHECKBOXES	Specifies the bit map for the check-box or radio-button check mark.
SBMP_CHILDSYSMENU	Specifies the bit map for the smaller version of the system-menu bit map; used in child windows.
SBMP_CHILDSYSMENUDEP	Same as SBMP_CHILDSYSMENU but indicates that the system menu is selected.
SBMP_COMBODOWN	Specifies the bit map for the downward pointing arrow in a drop-down combination box.
SBMP_MAXBUTTON	Specifies the bit map for the maximize button.
SBMP_MENUATTACHED	Specifies the bit map for the symbol used to indicate that a menu item has an attached, hierarchical menu.
SBMP_MENUCHECK	Specifies the bit map for the menu check mark.
SBMP_MINBUTTON	Specifies the bit map for the minimize button.
SBMP_OLD_CHILDSYSMENU	Same as SBM_CHILDSYSMENU. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_MAXBUTTON	Same as SBM_MAXBUTTON. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_MINBUTTON	Same as SBM_MINBUTTON. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_RESTOREBUTTON	Same as SBM_RESTOREBUTTON. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_SBDNARROW	Same as SBM_SBDNARROW. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_SBLFARROW	Same as SBM_SBLFARROW. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_SBRGARROW	Same as SBM_SBRGARROW. (For compatibility with previous versions of the OS/2 operating system.)
SBMP_OLD_SBUPARROW	Same as SBM_SBUPARROW. (For compatibility with previous versions of the OS/2 operating system.)

Table 26-3 (Page 2 of 2). Standard System Bit Maps	
Bit Map	**Description**
SBMP_PROGRAM	Specifies the bit map for the symbol that File Manager uses to indicate that a file is an executable program.
SBMP_RESTOREBUTTON	Specifies the bit map for the restore button.
SBMP_RESTOREBUTTONDEP	Same as SBMP_RESTOREBUTTON but indicates that the restore button is pressed.
SBMP_SBDNARROW	Specifies the bit map for the scroll-bar down arrow.
SBMP_SBDNARROWDEP	Same as SBMP_SBDNARROW but indicates that the scroll-bar down arrow is pressed.
SBMP_SBDNARROWDIS	Same as SBMP_SBDNARROW but indicates that the scroll-bar down arrow is disabled.
SBMP_SBLFARROW	Specifies the bit map for the scroll-bar left arrow.
SBMP_SBLFARROWDEP	Same as SBMP_SBLFARROW but indicates that the scroll-bar left arrow is pressed.
SBMP_SBMFARROWDIS	Same as SBMP_SBLFARROW but indicates that the scroll-bar left arrow is disabled.
SBMP_SBRGARROW	Specifies the bit map for the scroll-bar right arrow.
SBMP_SBRGARROWDEP	Same as SBMP_SBRGARROW but indicates that the scroll-bar right arrow is pressed.
SBMP_SBRGARROWDIS	Same as SBMP_SBRGARROW but indicates that the scroll-bar right arrow is disabled.
SBMP_SBUPARROW	Specifies the bit map for the scroll-bar up arrow.
SBMP_SBUPARROWDEP	Same as SBMP_SBUPARROW but indicates that the scroll-bar up arrow is pressed.
SBMP_SBUPARROWDIS	Same as SBMP_SBUPARROW but indicates that the scroll-bar up arrow is disabled.
SBMP_SIZEBOX	Specifies the bit map for the symbol that indicates an area of a window in which the user can click to resize the window.
SBMP_SYSMENU	Specifies the bit map for the system menu.
SBMP_TREEMINUS	Specifies the bit map for the symbol that File Manager uses to indicate an empty entry in the directory tree.
SBMP_TREEPLUS	Specifies the bit map for the symbol that File Manager uses to indicate that an entry in the directory tree contains more files.

Using Mouse Pointers and Icons

This section explains how to perform the following tasks:

- Save the current mouse pointer.
- Change the mouse pointer.
- Restore the original mouse pointer.

Changing the Mouse Pointer

Once you create or load a mouse pointer, you can change its shape by calling the WinSetPointer function. Following are three typical situations in which an application changes the shape of the mouse pointer:

- When an application receives a WM_MOUSEMOVE message, there is an opportunity to change the mouse pointer based on its location in the window. If you want the standard arrow pointer, pass this message on to the WinDefWindowProc function.
- When an application is about to start a time-consuming process during which it will not accept user input, the application displays the *system-wait* mouse pointer (SPTR_WAIT). Upon finishing the process, the application resets the mouse pointer to its former shape.

  The following code fragment shows how to save the current mouse pointer, set the hourglass pointer, and restore the original mouse pointer. Notice that the hourglass pointer also is saved in a global variable so that the application can return it when responding to a WM_MOUSEMOVE message during a time-consuming process.

```
HPOINTER hptrOld, hptrWait, hptrCurrent;

/* Get the current pointer.                    */
hptrOld = WinQueryPointer(HWND_DESKTOP);

/* Get the wait mouse pointer.                 */
hptrWait = WinQuerySysPointer(HWND_DESKTOP,
    SPTR_WAIT, FALSE);

/* Save the wait pointer to use in WM_MOUSEMOVE processing.*/
hptrCurrent = hptrWait;

/* Set the mouse pointer to the wait pointer. */
WinSetPointer(HWND_DESKTOP, hptrWait);

/*
 * Do a time-consuming operation, then restore the
 * original mouse pointer.
 */
WinSetPointer(HWND_DESKTOP, hptrOld);
```

- When an application needs to indicate its current operational mode, it changes the pointer shape. For example, a paint program with a palette of drawing tools should change the pointer shape to indicate which drawing tool is in use currently.

Summary

Following are the OS/2 functions and structure used with mouse pointers, icons, and bit maps.

Table 26-4 (Page 1 of 2). Pointer and Bit Map Functions

Function name	Description
WinCreatePointer	Creates a pointer from a bit map.

Table 26-4 (Page 2 of 2). Pointer and Bit Map Functions	
Function name	**Description**
WinCreatePointerIndirect	Creates a colored pointer or icon from a bit map.
WinDestroyPointer	Destroys a pointer or an icon.
WinDrawBitmaps	Draws a bit map using the current image colors and mixes.
WinDrawPointer	Draws a pointer.
WinGetSysBitmap	Returns a handle to one of the standard bit maps provided by the system.
WinLoadPointer	Loads a pointer from a resource file into the system.
WinQueryPointer	Returns the pointer handle for **DeskTop**.
WinQueryPointerInfo	Returns pointer information.
WinQueryPointerPos	Returns the pointer position.
WinQuerySysPointer	Returns the handle of the system pointer.
WinSetPointer	Sets the handle of the Desktop pointer.
WinSetPointerPos	Sets the pointer position.
WinShowPointer	Adjusts the pointer display level to show or hide a pointer.

Table 26-5. Pointer Structure	
Structure	**Description**
POINTERINFO	Pointer information structure.

Chapter 27. Cursors

A *cursor* is a rectangle that can be shown at any location in a window, indicating where the user's next interaction with items on the screen will happen. This chapter describes how to create and use cursors in your PM applications.

About Cursors

Only one cursor appears on the screen at a time—either marking the text-insertion point (a *text cursor*) or indicating which items the user can interact with from the keyboard (a *selection cursor*). For example, when an entry field has the keyboard focus, it displays a blinking vertical bar to show the text-insertion point; however, when a button has the keyboard focus, the cursor appears as a halftone rectangle the size of the button. The operating system draws and blinks the cursor, freeing the application from handling these details. Notice that the cursor has no direct relationship with the mouse pointer.

Cursor Creation and Destruction

The system can use only one cursor at a time, so windows must create and destroy cursors as each windows gains and loses the keyboard focus. If an application attempts to use more than one cursor at a time, the results can be unpredictable and might affect other applications.

An application creates a cursor by calling WinCreateCursor. Generally, this is done when a window gains the keyboard focus. The application specifies the window in which to display the cursor, whether it be the desktop window, an application window, or a control window. An application destroys a cursor by calling WinDestroyCursor— when the specified window loses the keyboard focus for example.

Position and Size

An application can set the position (in window coordinates) of an existing cursor by calling WinCreateCursor, specifying the CURSOR_SETPOS flag. The cursor width is usually 0 (nominal border width is used) for text-insertion cursors. This is preferable to a value of 1, since such a fine width is almost invisible on a high-resolution monitor. The cursor width also can be related to the window size—for example, when a button control uses a dotted-line cursor around the button text to indicate focus. To change the cursor size, the application must destroy the current cursor and create a new one of the desired size.

Other Cursor Characteristics

An application uses the WinCreateCursor function to specify information about the cursor rectangle and the clipping rectangle. WinCreateCursor specifies whether the cursor rectangle should be filled, framed, blinking, or halftone. In addition, the function specifies the clipping rectangle, in window coordinates, that controls the cursor clipping region. Probably the most efficient strategy is for the application to specify NULL, which causes the rectangle to clip the cursor to the window rectangle.

Cursor Visibility

An application can use the WinShowCursor function to show or hide a cursor. The operating system maintains a *show level* for the cursor: when the cursor is visible, the its show level is zero; each time the cursor is hidden, its show level is incremented; each time the cursor is shown, its show level is decremented. The show:hide relationship is 1:1, so the show level cannot drop below zero. When first creating a cursor, an application should show the cursor because the application creates the cursor with a show level of 1.

The operating system automatically hides the cursor when the application calls WinBeginPaint; it shows the cursor when the application calls WinEndPaint. Therefore, there is no conflict with the cursor during WM_PAINT processing.

Using Cursors

This section explains how to perform the following tasks:

- Create and destroy a cursor.
- Respond to a WM_SETFOCUS message.

Creating and Destroying a Cursor

The following code fragment shows how an application should respond to a WM_SETFOCUS message when using a cursor in a particular window:

```
LONG   curXPos,curYPos,curWidth,curHeight;

case WM_SETFOCUS:
    if (SHORT1FROMMP(mp2)) {

        /* Gain the focus. */
        WinCreateCursor(hwnd, curXPos, curYPos, curWidth, curHeight,
            CURSOR_SOLID | CURSOR_FLASH, (PRECTL) NULL);
        WinShowCursor(hwnd, TRUE);
    }
    else {

        /* Lose the focus. */
        WinDestroyCursor(hwnd);
    }

    return 0;
```

Figure 27-1. Response to a WM_SETFOCUS message

Summary

Following are the OS/2 functions and structure used with cursors:

Table 27-1. Cursor Functions

Function name	Description
WinCreateCursor	Used to create, set the size of, and move the cursor around the screen.
WinDestroyCursor	Destroys the current cursor if it belongs to the specified window.
WinQueryCursorInfo	Obtains information about any current cursor.
WinShowCursor	Shows or hides the cursor associated with a specified window.

Table 27-2. Cursor Structure

Structure name	Description
CURSORINFO	Cursor information structure.

Chapter 28. Painting and Drawing

This chapter describes presentation spaces, device contexts, and window regions, explaining how a PM application uses them for painting and drawing in windows.

About Painting and Drawing

An application typically maintains an internal representation of the data that it is manipulating. The information displayed in a screen, window, or printed copy is a visual representation of some portion of that data. This chapter introduces the concepts and strategies necessary to make your PM application function smoothly and cooperatively in the OS/2 display environment.

Presentation Spaces and Device Contexts

A *presentation space* is a data structure, maintained by the operating system, that describes the drawing environment for an application. An application can create and hold several presentation spaces, each describing a different drawing environment. All drawing in a PM application must be directed to a presentation space.

Normally each presentation space is associated with a *device context* that describes the physical device where graphics commands are displayed. The device context translates graphics commands made to the presentation space into commands that enable the physical device to display information. Typical device contexts are the screen, printers and plotters, and off-screen memory bit maps.

Figure 28-1 shows how graphics commands from an application go through a presentation space, to a device context, and then to the physical device.

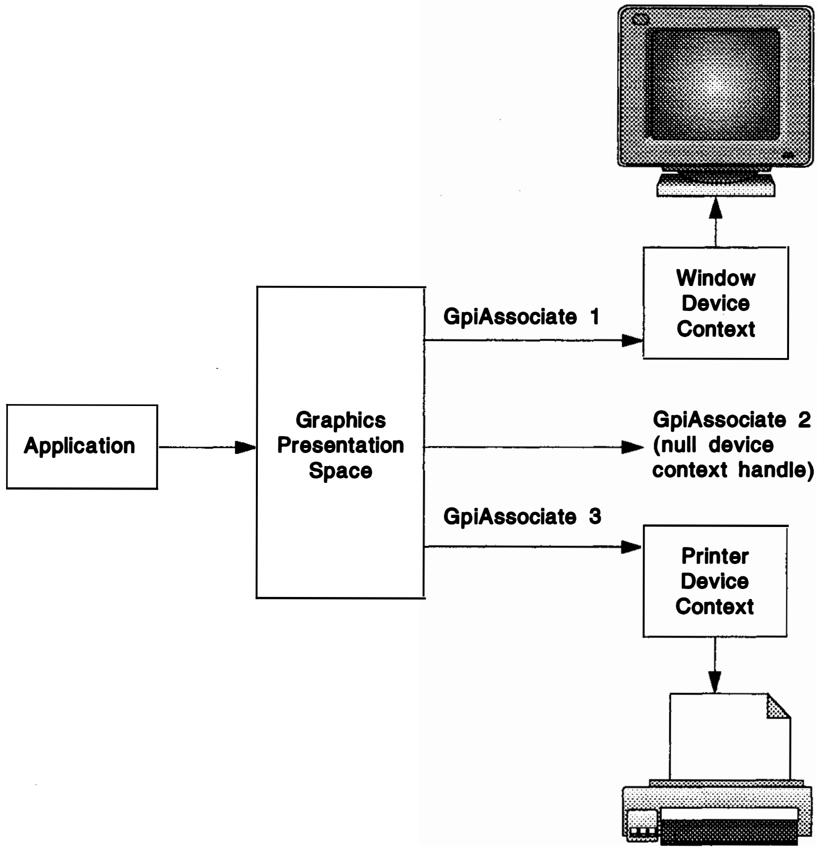


Figure 28-1. Application's Flow of Graphics Commands

By creating presentation spaces and associating them with particular device contexts, an application can control where its graphics output appears. Typically, a presentation space and device context isolate the application from the physical details of displaying graphics, so the same graphics commands can be used for many types of displays. This virtualization of output can reduce the amount of display code an application must include to support multiple output devices.

This chapter describes how an application sets up its presentation spaces and device contexts before drawing, and how to use window-drawing functions. Refer to the *OS/2 2.0 Programming Guide, Volume III—Graphics Programming Interface* for the graphics functions available to PM applications.

Window Regions

A window and its associated presentation space have three regions that control where drawing takes place in the window. These regions ensure that the application does not draw outside the boundaries of the window or intrude into the space of an overlapping window.

Table 28-1. Window Regions

Region	Description
Update Region	This region represents the area of the window that needs to be redrawn. This region changes when overlapping windows change their z-order or when an application explicitly adds an area to the update region to force a window to be painted.
Clip Region	This region and the visible region determine where drawing takes place. Applications can change the clip region to limit drawing to a particular portion of a window. Typically, a presentation space is created with a clip region equal to NULL, which makes this region equivalent to the update region.
Visible Region	This region and the clip region determine where drawing takes place. The system changes the visible region to represent the portion of a window that is visible. Typically, the visible region is used to mask out overlapping windows. When an application calls the WinBeginPaint function in response to a WM_PAINT message, the system sets the visible region to the intersection of the visible region and the update region to produce a new visible region. Applications cannot change the visible region directly.

Whenever drawing occurs in a window's presentation space, the output is clipped to the intersection of the visible region and clip region. Figure 28-2 shows how the intersection of the visible region and the clip region of a window that is behind another window prevents the drawing in the back window from intruding into the front window. The clip region includes the overlapped part of the back window, but the visible region excludes that portion of the back window. The system maintains the visible region to protect other windows on the screen; the application maintains the clip region to specify the portion of the window in which it draws. Together, these two regions provide safe and controllable clipping.

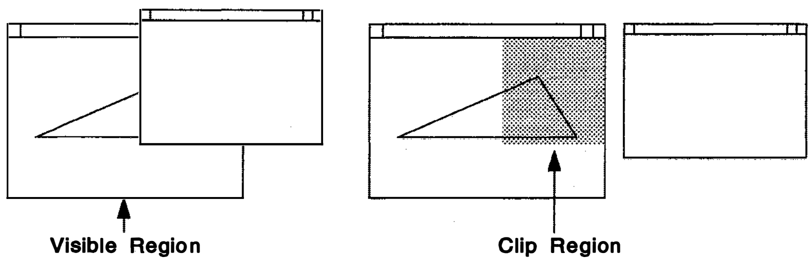


Figure 28-2. Clip Region and Visible Region of a Window's Presentation Space

To further control drawing, both the system and the application manipulate the update region. For example, if the windows shown in Figure 28-2 switch positions front to back, several changes occur in the regions of both windows. The system adds the lower-right corner of the new front window to that window's visible region. The system also adds that corner area to the window's update region.

Window Styles for Painting

Most of the styles relating to window drawing can be set either for the window class (CS_ prefix) or for an individual window (WS_ prefix). The styles described in this section control how the system manipulates the window's regions and how the window is notified when it must be painted or redrawn.

WS_CLIPCHILDREN, CS_CLIPCHILDREN

All the windows with this style are excluded from their parent's visible region. This style protects windows but increases the amount of time necessary to calculate the parent's visible region. This style normally is not necessary, because if the parent and child windows overlap and both are invalidated, the parent window is drawn before the child window. If the child window is invalidated independently from its parent window, only the child window is redrawn. If the update region of the parent window does not intersect the child window, drawing the parent window does not disturb the child window.

WS_CLIPSIBLINGS, CS_CLIPSIBLINGS

Windows with this style are excluded from the visible region of sibling windows. This style protects windows with the same parent from being drawn accidentally, but increases the amount of time necessary to calculate the visible region. This style is appropriate for sibling windows that overlap.

WS_PARENTCLIP, CS_PARENTCLIP

The visible region for a window with this style is the same as the visible region of the parent window. This style simplifies the calculation of the visible region but is potentially hazardous, because the parent window's visible region usually is larger than the child window. Windows with this style should not draw outside their boundaries.

WS_SAVEBITS, CS_SAVEBITS

The system saves the bits beneath a window with this style when the window is displayed. When the window moves or is hidden, the system simply restores the uncovered bits. This operation can consume a great deal of memory; it is recommended only for transient windows such as menus and dialog boxes—not for main application windows. This style also is inappropriate for windows that are updated dynamically, such as clocks.

WS_SYNCPAINT, CS_SYNCPAINT

Windows that have these styles receive WM_PAINT messages as soon as their update regions contain something; they are updated immediately (synchronously).

CS_SIZEREDRAW

A window with this class style receives a WM_PAINT message; the window is completely invalidated whenever it is resized, even if it is made smaller. (Typically, only the uncovered area of a window is invalidated when a window is resized.) This class style is useful when an application scales graphics to fill the current window.

Strategies for Painting and Drawing

A PM application shares the screen with other windows and applications; therefore, painting and drawing must not interfere with those other applications and windows. When you follow these strategies, your application can coexist with other applications and still take full advantage of the graphics capabilities of the operating system.

Drawing in a Window

Ideally, all drawing in a window occurs as a result of an application's processing a WM_PAINT message. Applications maintain an internal representation of what must be displayed in the window, such as text or a linked list of graphics objects, and use the WM_PAINT message as a cue to display a visual representation of that data in the window.

To route all display output through the WM_PAINT message, an application must not draw on the screen at the time its data changes. Instead, it must update the internal representation of the data and call the WinInvalidateRect or WinInvalidateRegion functions to invalidate the portion of the window that must be redrawn. Sometimes it is much more efficient to draw directly in a window without relying on the WM_PAINT message—for example, when drawing and redrawing an object for a user who is using the mouse to drag or size the object.

If a window has the WS_SYNCPAINT or CS_SYNCPAINT style, invalidating a portion of the window causes a WM_PAINT message to be sent to the window immediately. Essentially, sending a message is like making a function call; the actions corresponding to the WM_PAINT message are carried out before the call that caused the invalidation returns—that is to say, the painting is synchronous.

If the window does not have the WS_SYNCPAINT or CS_SYNCPAINT style, invalidating a portion of the window causes the invalidated region to be added to the window's update region. The next time the application calls the WinGetMsg or WinPeekMsg functions, the application is sent a WM_PAINT message. If there are many messages in the queue, the painting occurs after the invalidation—that is, the painting is asynchronous. A WM_PAINT message is not posted to the queue in this case, so all invalidation operations since the last WM_PAINT message are consolidated into a single WM_PAINT message the next time the application has no messages in the queue.

There are advantages to both synchronous and asynchronous painting. Windows that have simple painting functions should be painted synchronously. Most of the system-defined control windows, such as buttons and frame controls, are painted synchronously because they can be painted quickly without interfering with the responsiveness of the program. Windows that require more time-consuming painting operations should be painted asynchronously so that the painting can be initiated only when there are no other pending messages that might otherwise be blocked while waiting for the window to be painted. Also, a window that uses an incremental approach to invalidating small portions of itself usually should allow those operations to consolidate into a single asynchronous WM_PAINT message, rather than a series of synchronous WM_PAINT messages.

If necessary, an application can call the WinUpdateWindow function to cause an asynchronous window to update itself without going through the event loop. WinUpdateWindow sends a WM_PAINT message directly to the window if the window's update region is not empty.

The WM_PAINT Message

A window receives a WM_PAINT message whenever its update region is not NULL. A window procedure responds to a WM_PAINT message by calling the WinBeginPaint function, drawing to fill in the update areas, then calling the WinEndPaint function.

The WinBeginPaint function returns a handle to a presentation space that is associated with the device context for the window and that has a visible region equal to the intersection of the window's update region and its visible region. This means that only those portions of the window that need to be redrawn are drawn. Attempts to draw outside this region are clipped and do not appear on the screen.

If the application maintains its own presentation space for the window, it can pass the handle of that presentation space to WinBeginPaint, which modifies the visible region of the presentation space and passes the presentation-space handle back to the caller. If the application does not have its own presentation space, it can pass a NULL presentation-space handle and the system will return a cached-micro presentation space for the window. In either case, the application can use the presentation space to draw in the window.

The WinBeginPaint function takes a pointer to a RECTL structure, filling in this structure with the coordinates of the rectangle that encloses the area to be updated. The application can use this rectangle to optimize drawing, by drawing only those portions of the window that intersect with the rectangle. If an application passes a NULL pointer for the rectangle argument, the application draws the entire window and relies on the clipping mechanism to filter out the unneeded areas.

After the WinBeginPaint function sets the update region of a window to NULL, the application does the necessary drawing to fill the update areas. If an application handles a WM_PAINT message and does not call WinBeginPaint, or otherwise empty the update region, the application continues to receive WM_PAINT messages as long as the update region is not empty.

After the application finishes drawing, it calls the WinEndPaint function to restore the presentation space to its former state. When a cached-micro presentation space is returned by WinBeginPaint, the presentation space is returned to the system for reuse. If the application supplies its own presentation space to WinBeginPaint, the presentation space is restored to its previous state.

Drawing the Minimized View

When an application creates a standard frame window, it has the option of specifying an icon that the system uses to represent the application in its minimized state. Typically, if an icon is supplied, the system draws it in the minimized window and labels it with the name of the window. If the application does not specify the FS_ICON style for the window, the window receives a WM_PAINT message when it is minimized. The code in the window procedure that handles the WM_PAINT message can determine whether the frame window currently is minimized and draw accordingly. Notice that because the WS_MINIMIZED style is relevant only for the frame window, and not for the client window, the window procedure checks the frame window rather than the client window.

The following code fragment shows how to draw a window in both the minimized and normal states:

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2)
{
    HPS hps;
    RECTL rcl;
    ULONG flStyle;

    switch (msg) {
        case WM_PAINT:
            hps = WinBeginPaint(hwnd, (HPS) NULL, &rcl);

            /*
             * Check whether the frame window (client's parent window)
             * is minimized.
             */

            flStyle = WinQueryWindowULong(WinQueryWindow(hwnd,
                QW_PARENT), QWL_STYLE);

            if (flStyle & WS_MINIMIZED) {
                .
                .    /* Paint the minimized state. */
                .
            }
            else {
                .
                .    /* Paint the normal state.    */
                .
            }
            WinEndPaint(hps);
            return 0;
    }
}
```

Drawing Without the WM_PAINT Message

An application can draw in a window's presentation space without having received a WM_PAINT message. As long as there is a presentation space for the window, an application can draw into the presentation space and avoid intruding into other windows or the desktop. Applications that draw without using the WM_PAINT message typically call the WinGetPS function to obtain a cached-micro presentation space for the window and call the WinReleasePS function when they have finished drawing. An application also can use any of the other types of presentation spaces described in the following sections.

Three Types of Presentation Spaces

All drawing must take place within a presentation space.

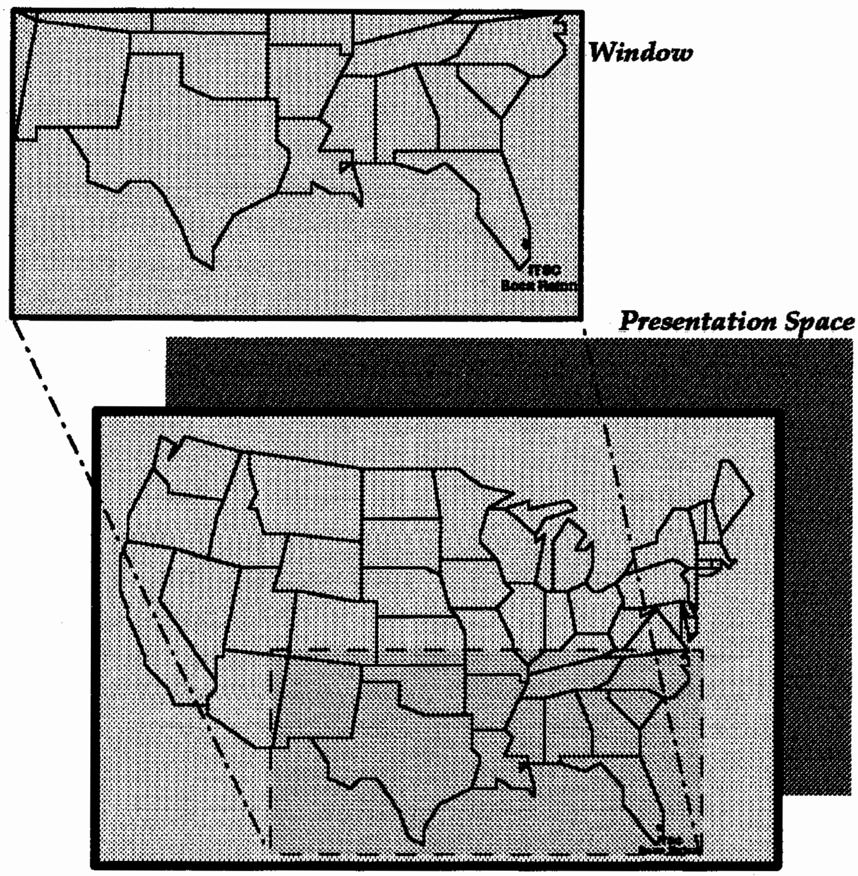


Figure 28-3. Presentation Space versus Window

The operating system provides three types of presentation spaces for drawing: normal, micro, and cached-micro presentation spaces.

The *normal presentation space* provides the most functionality, allowing access to all the graphics functions of the operating system and enabling the application to draw to all device types. The normal presentation space is more difficult to use than the other two kinds of presentation spaces and it uses more memory. It is created by using the GpiCreatePS function and is destroyed by using the GpiDestroyPS function.

The *micro presentation space* allows access to only a subset of the operating system graphics functions, but it uses less memory and is faster than a normal presentation space. The micro presentation space also enables the application to draw to all device types. It is created by using the GpiCreatePS function and destroyed by using the GpiDestroyPS function.

The *cached-micro presentation space* provides the least functionality of the three kinds of presentation spaces, but it is the most efficient and easiest to use. The cached-micro presentation space draws only to the screen. It is created and destroyed by using either the WinBeginPaint and WinEndPaint functions or the WinGetPS and WinReleasePS functions.

The following sections describe each of the types of presentation spaces, in detail, and suggest strategies for using each type in an application. All three kinds of presentation spaces can be used in a single application. Some windows, especially if they never will be printed, are best served by cached-micro presentation spaces. Other windows might require the more flexible services of micro or normal presentation spaces.

Normal Presentation Spaces

The normal presentation space supports the full power of the operating system graphics, including retained graphics. The primary advantages of a normal presentation space over the other two presentation-space types are its support of all graphics functions and its ability to be associated with many kinds of device contexts.

A normal presentation space can be associated with many different device contexts. Typically, this means that an application creates a normal presentation space and associates it with a window device context for screen display. When the user asks to print, the application associates the same presentation space with a printer device context. Later, the application can reassociate the presentation space with the window device context. A presentation space can be associated with only one device context at a time, but the normal presentation space enables the application to change the device context whenever necessary.

Figure 28-4 shows how an application typically routes graphics through one normal presentation space into another device context:

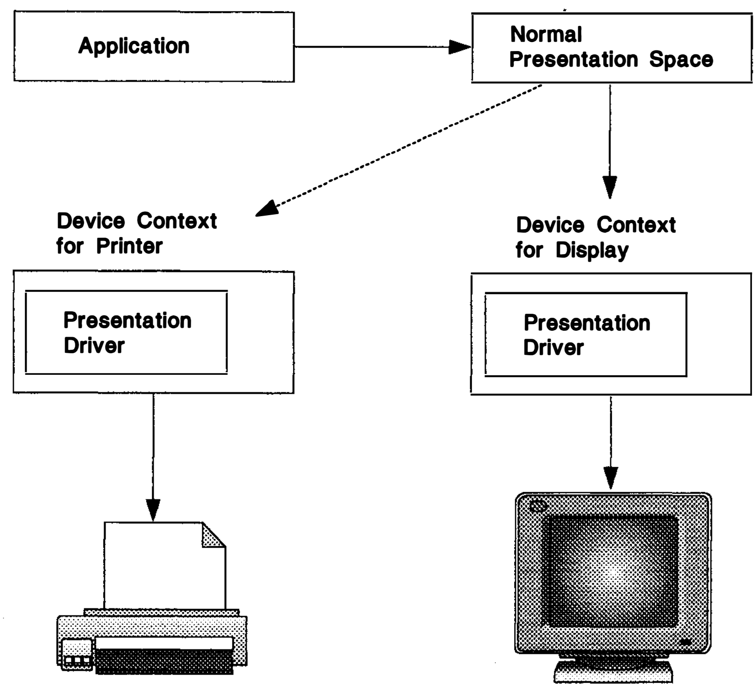


Figure 28-4. Normal Presentation Space

When creating a normal presentation space, an application can associate it with a device context or defer the association to a later time. The GpiAssociate function associates a device context with a normal presentation space after the presentation space has been created. An application typically associates the normal presentation space with a device context when calling the GpiCreatePS function and, later, associates the presentation space with a different device context by calling GpiAssociate. To obtain a device context for a window, call the WinOpenWindowDC function. To obtain a device context for a device other than the screen, call the DevOpenDC function.

An application typically creates a normal presentation space during initialization and uses it until termination. Each time the application receives a WM_PAINT message, it passes the handle of the normal presentation space as an argument to WinBeginPaint; this prevents the system from returning a cached-micro presentation space. The system modifies the visible region of the supplied normal presentation space and returns the presentation space to the application. This method enables the application to use the same presentation space for all the drawing in a specified window.

Normal presentation spaces created using GpiCreatePS must be destroyed by calling GpiDestroyPS before the application terminates. Do not call WinReleasePS to release a presentation space obtained using GpiCreatePS. Before terminating, applications also must use DevCloseDC to close any device contexts opened using DevOpenDC. No action is necessary for device contexts obtained using WinOpenWindowDC, because the system automatically closes these device contexts when destroying the associated windows.

Micro Presentation Spaces

The primary advantage of a micro presentation space over a cached-micro presentation space is that it can be used for printing as well as painting in a window. An application that uses a micro presentation space must explicitly associate it with a device context. This makes the micro presentation space useful for painting to a printer, a plotter, or an off-screen memory bit map.

A micro presentation space does not support the full set of OS/2 graphics functions. Unlike a normal presentation space, a micro presentation space does not support retained graphics.

An application that must display graphics or text in a window and print to a printer or plotter typically maintains two presentation spaces: one for the window and one for the printing device. Figure 28-5 shows how an application's graphics output can be routed through separate presentation spaces to produce a screen display and printed copy.

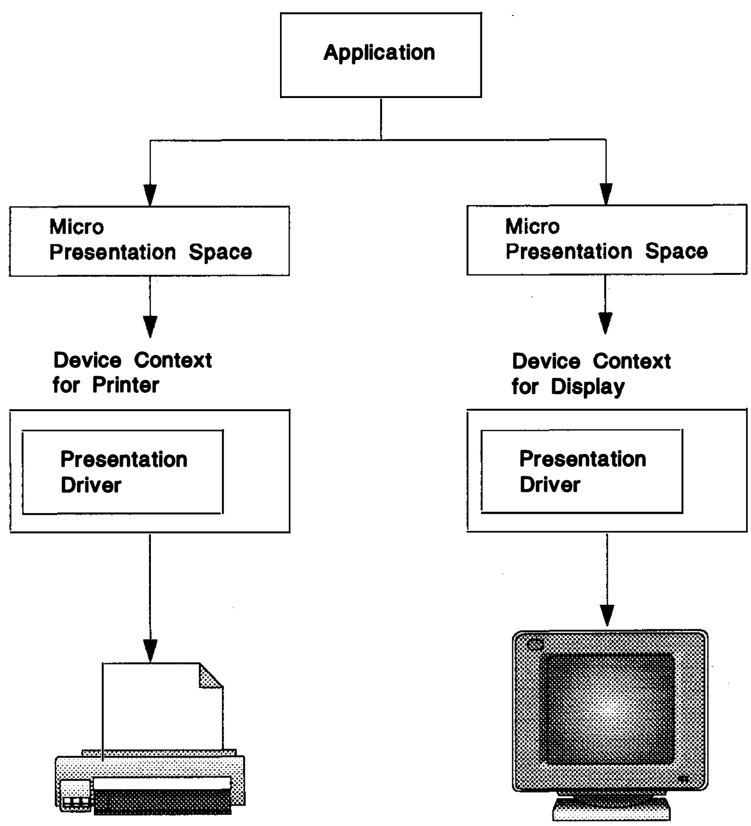


Figure 28-5. Micro Presentation Space

An application creates a micro presentation space by calling the GpiCreatePS function. A device context must be supplied at the time the micro presentation space is created. An application typically creates a device context and then a presentation space. The following code fragment demonstrates this by obtaining a device context for a window and associating it with a new micro presentation space:

```
hdc = WinOpenWindowDC(...);
hps = GpiCreatePS(..., hdc, ..., GPIA_ASSOC);
```

To create a micro presentation space for a device other than the screen, replace the call to the WinOpenWindowDC function with a call to the DevOpenDC function, which obtains a device context for a device other than the screen. Then the device context that is obtained by this call can be used as an argument to GpiCreatePS.

An application typically creates a micro presentation space during initialization and uses it until termination. Each time the application receives a WM_PAINT message, it should pass the handle of the micro presentation space as an argument to the WinBeginPaint function; this prevents the system from returning a cached-micro presentation space. The system modifies the visible region of the supplied micro presentation space and returns the presentation space to the application. This method enables the application to use the same presentation space for all drawing in a specified window.

Micro presentation spaces created by using GpiCreatePS should be destroyed by calling GpiDestroyPS before the application terminates. Do not call the WinReleasePS function to release a presentation space obtained by using GpiCreatePS. Before terminating, applications must use the DevCloseDC function to close any device contexts opened using the DevOpenDC function. No action is necessary for device contexts obtained using WinOpenWindowDC, because the system automatically closes these device contexts when destroying the associated windows.

Cached-Micro Presentation Spaces

The cached-micro presentation space provides the simplest and most efficient drawing environment. It can be used only for drawing on the screen, typically in the context of a window. It is most appropriate for application tasks that require simple window-drawing functions that will not be printed. Cached-micro presentation spaces do not support retained graphics.

After an application draws to a cached-micro presentation space, the drawing commands are routed through an implied device context to the current display. The application does not require information about the actual device context, because the device context is assumed to be the display. This process makes cached-micro presentation spaces easy for applications to use.

The following code fragment illustrates this process:

```
HPS    hps;

    case WM_PAINT:
        hps = WinBeginPaint(hwnd,NULL,NULL);

        /*
         * Use PS.
         */

        WinEndPaint (hps);
```

or

```
HPS    hps;

    case WM_PAINT:

        hps = WinGetPS(hwnd);

        /*
        * Use PS.
        */

        WinReleasePS(hps);
```

There are two common strategies for using cached-micro presentation spaces in an application. The simplest strategy is to call the WinBeginPaint function during the WM_PAINT message, use the resulting cached-micro presentation space to draw in the window, then return the presentation space to the system by calling the WinEndPaint function. By using this method, the application interacts with the presentation space only when drawing in the presentation space. This method is most appropriate for simple drawing. A disadvantage of this method is that the application must set up any special attributes for the presentation space, such as line color and font, each time a new presentation space is obtained.

A second strategy is for the application to allocate a cached-micro presentation space during initialization, by calling the WinGetPS function and saving the resulting presentation-space handle in a static variable. Then the application can set attributes in the presentation space that exist for the life of the program. The presentation-space handle can be used as an argument to the WinBeginPaint function each time the window gets a WM_PAINT message; the system modifies the visible region and returns the presentation space to the application with its attributes intact. This strategy is appropriate for applications that need to customize their window-drawing attributes.

A presentation space that is obtained by calling the WinGetPS function must be released by calling WinReleasePS when the application has finished using it, typically during program termination. A presentation space that is obtained by calling WinBeginPaint must be released by calling WinEndPaint, typically as the last part of processing a WM_PAINT message.

Summary

Following are the OS/2 functions used with presentation spaces, device contexts, and window regions.

Table 28-2. Presentation Space, Device Context, and Window Region Functions

Function Name	Description
DevCloseDC	Closes a device context.
DevOpenDC	Opens a device context.
GpiAssociate	Associates a graphics presentation space with, or disassociates it from, a device context.
GpiCreatePs	Creates a presentation space
GpiDestroyPS	Destroys a presentation space.
WinBeginPaint	Obtains a presentation space whose associated update region is set to draw in a specified window.
WinEnableWindowUpdate	Sets the visibility state for subsequent drawing.
WinEndPaint	Indicates that the redrawing of a window is complete.
WinExcludeUpdateRegion	Subtracts the update region of a window from the clipping region of a presentation space.
WinGetClipPS	Obtains a clipped cache presentation space.
WinGetPS	Gets a cache presentation space.
WinGetScreenPS	Returns a presentation space that can be used for drawing anywhere on the screen.
WinInvalidateRect	Adds a rectangle to a window's update region.
WinLockVisRegions	Locks or unlocks the visible regions or all the windows
WinLockWindowUpdate	Disables or enables output to a window and its descendants.
WinOpenWindowDC	Opens a device context for a window.
WinQueryUpdateRect	Returns the rectangle that bounds the update region of a specified window.
WinQueryUpdateRegion	Obtains an update region of a window.
WinQueryWindowDC	Returns the device context for a given window.
WinReleasePS	Releases a cache presentation space obtained using the WinGetPS or WinGetScreenPS calls.
WinValidateRect	Subtracts a rectangle from the update region of an asynchronous paint window, marking that part of the window as visually valid.
WinValidateRegion	Subtract a a region from the update region of an asynchronous paint window, marking that part of the widow as visually valid.
WinWindowFromDC	Returns the handle of the window corresponding to a particular device context.

Chapter 29. Drawing in Windows

This chapter describes, at a high level, the functions specifically intended for drawing in PM windows. For information on the complete set of drawing functions, see the *OS/2 2.0 Programming Guide, Volume III*—Graphics Programming Interface.

About Window-Drawing Functions

The functionality of the PM window-drawing functions overlaps that of similar Gpi drawing functions in OS/2 These window-drawing functions are less general than the Gpi functions and are somewhat easier to use, but they also offer fewer capabilities than the complete set of Gpi functions. Programmers requiring optimum functionality should use the Gpi functions.

Points

All drawing in a window takes place in the context of the window's coordinate system. Locations of points in the window are described by POINTL structures, which contain an x and a y coordinate for the point. The lower-left corner of a window always has the coordinates (0,0).

The WinMapWindowPoints function converts the coordinates of points from one window-coordinate space to those of another window-coordinate space. If one of the specified windows is HWND_DESKTOP, the function uses screen coordinates. This function is useful for converting window coordinates to screen coordinates or the other way around.

Rectangles

Locations of window rectangles are described by RECTL structures, which contain the coordinates of two points that define the lower-left and upper-right corners of the rectangle. An empty rectangle is one that has no area: either its right coordinate is less than or equal to its left coordinate, or its top coordinate is less than or equal to its bottom coordinate.

There are two types of rectangles in OS/2: inclusive-inclusive and inclusive-exclusive. In inclusive-exclusive rectangles the lower-left coordinate of the rectangle is included within the rectangle area, while the upper-right coordinate is excluded from the rectangle area. In an inclusive-inclusive rectangle, both the lower-left and upper-right coordinates are included in the rectangle. Figure 29-1 shows both types of rectangles:

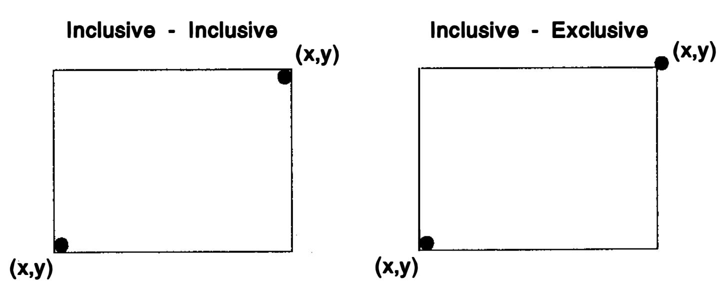


Figure 29-1. Types of Rectangles

In general, graphics operations involving device coordinates (such as regions, bit maps and bit blits, and window management) use inclusive-exclusive rectangles. All other graphics operations, such as Gpi functions that define paths, use inclusive-inclusive rectangles.

Using Window-Drawing Functions

This section explains how to use drawing functions to fill (paint) a rectangle with color, scroll the contents of a window, draw bit maps and text, and determine the dimensions of a rectangle.

Working with Points and Rectangles

The operating system includes functions for manipulating rectangles, many of which change the rectangle coordinates. Other functions draw in a presentation space, using a rectangle to position the drawing operation.

The rest of the rectangle functions are mathematical and do not draw. They are used to manipulate and combine rectangles to produce new rectangles that you then can use in drawing operations.

Determining the Dimensions of a Rectangle

You can calculate the dimensions of an inclusive-exclusive rectangle as follows:

```
cx = rcl.xRight - rcl.xLeft; /* width  */
cy = rcl.yTop - rcl.yBottom; /* height */
```

You can calculate the dimensions of an inclusive-inclusive rectangle as follows:

```
cx = (rcl.xRight - rcl.xLeft) + 1; /* width  */
cy = (rcl.yTop - rcl.yBottom) + 1; /* height */
```

Filling a Rectangle

The WinFillRect function fills (paints) a rectangle with a specified color. For example, to fill an entire window with blue in response to a WM_PAINT message, you could use the following code fragment, which is taken from a window procedure:

```
HPS    hps;
RECTL  rcl;


case WM_PAINT:
    hps = WinBeginPaint(hwnd, (HPS) NULL, (PRECTL) NULL);
    WinQueryWindowRect(hwnd, &rcl);
    WinFillRect(hps, &rcl, CLR_BLUE);
    WinEndPaint(hps);
    return 0;
```

A more efficient way of painting a client window is to pass a rectangle to the WinBeginPaint function. The rectangle is set to the coordinates of the rectangle that encloses the update region of the window. Drawing in this rectangle updates the window, which can make drawing faster if only a small portion of the window needs to be painted. This method is shown in the following code fragment. Notice that WinFillRect uses the presentation space and a rectangle defined in window coordinates to guide the paint operation.

```
HPS    hps;
RECTL  rcl;

case WM_PAINT:
    hps = WinBeginPaint(hwnd, (HPS) NULL, &rcl);
    WinFillRect(hps, &rcl, CLR_BLUE);
    WinEndPaint(hps);
    return 0;
```

You could draw the entire window during the WM_PAINT message, but the graphics output would be clipped to the update region.

The default method of indicating that a particular portion of a window has been selected is using the WinInvertRect function to invert the rectangle's bits.

Scrolling the Contents of a Window

An application typically responds to a click in a scroll bar by scrolling the contents of the window. This operation has three parts. First, the application changes its internal data-representation state to show what portion of the image must now be in the window. Next, the application moves the current image in the window. Finally, the application draws in the area that has been uncovered by the scrolling operation.

For example, a simple text editor might display a small portion of several pages of text in a window. When the user clicks the Down arrow of the vertical scroll bar, the application moves all the text up one line and displays the next line at the bottom of the window.

This clicking also causes a message to be sent to the client window of the frame window that owns the scroll bar. The application responds to this message by changing its internal data-representation state to show which line of text is topmost in the window, scrolling the text in the window up one line, and drawing the new line at the bottom of the window. There normally is no need to completely redraw the entire window, because the scrolled portion of the image remains valid.

You can use the WinScrollWindow function to scroll the contents of your application windows. WinScrollWindow scrolls a specified rectangular area of the window by a specified x and y offset (in window coordinates). If you set the SW_INVALIDATERGN flag for this function, the areas you uncover by scrolling are added to the window's update region automatically, causing a WM_PAINT message for the areas to be sent to the window.

For example, as used in the simple text editor described previously, the following call scrolls the text up one line (assuming that the *iVScrollInc* parameter specifies the height of the current font) and adds the uncovered area at the bottom of the window to the update region:

```
HWND hwnd;
LONG iVScrollInc;

/* Scroll, adding a new area to the update region.          */

WinScrollWindow(hwnd,  /* Window handle                    */
    0,                 /* x displacement                   */
    -(iVScrollInc),    /* y displacement                   */
    (PRECTL) NULL,     /* Scroll rectangle is entire window */
    (PRECTL) NULL,     /* Clip rectangle is entire window  */
    (HRGN) NULL,       /* Update region                    */
    (PRECTL) NULL,     /* Update rectangle                 */
    SW_INVALIDATERGN); /* Scroll-window flag               */
```

When the uncovered area is added to the window's update region, a WM_PAINT message is sent to the window. Upon receiving the message, the window draws the line of text at the bottom of the window. If the window has the WS_SYNCPAINT style, the WM_PAINT message is sent to the window before WinScrollWindow returns.

To optimize scrolling speed for repeated scrolling operations, you can omit the SW_INVALIDATERGN flag from the call to WinScrollWindow, which prevents the function from adding the invalid region (uncovered by the scroll) to the window's update region. If you omit the SW_INVALIDATERGN flag, you must pass a region or rectangle to WinScrollWindow. The rectangle or region will contain the area that must be updated after scrolling.

Drawing a Bit Map

The WinDrawBitmap function draws a bit map, identified by a bit map handle, in a specified rectangle. This function enables you to reduce or enlarge the bit map from the source rectangle to the destination rectangle. WinDrawBitmap also can draw in several different copy modes, including using the OR operator to combine source and destination pels.

Drawing Text

There are many ways to draw text in a window in an OS/2 application. The simplest way is to use the WinDrawText function, which draws a single line of text in a specified rectangle, using a variety of alignment methods.

WinDrawText allows you to set a flag so that the function does not draw any text; instead, the function returns the number of characters in the string that will fit in the specified rectangle. For a section of running text, an application can alternate between computation and calls to WinDrawText to draw successive lines of text. When performing this kind of repetitive operation, you can set the DT_WORDBREAK flag in the WinDrawText function to put line breaks on word boundaries rather than between arbitrary characters.

Summary

Following are the OS/2 functions and structures used for drawing in windows.

Table 29-1. Window-Drawing Functions

Function Name	Description
WinCalcFrameRect	Calculates a client rectangle from a frame rectangle or a frame rectangle from a client rectangle.
WinCopyRect	Copies a rectangle from *prclSrc* to *prclDest*.
WinDrawBitmap	Draws a bit map using the current image colors and mixes.
WinDrawBorder	Draws the borders and interior of a rectangle.
WinDrawText	Draws a single line of formatted text into a specified rectangle.
WinEqualRect	Compares two rectangles for equality.
WinFillRect	Draws a filled rectangular area.
WinInflateRect	Expands a rectangle.
WinIntersectRect	Calculates the intersection of the two source rectangles and returns the result in the destination rectangle.
WinInvalidateRect	Adds a rectangle to a window's update region.
WinInvertRect	Inverts a rectangular area.
WinIsRectEmpty	Determines whether a rectangle is empty.
WinMakeRect	Converts points to graphics points.
WinMapWindowPoints	Maps points from dialog coordinates to window coordinates or from window coordinates to dialog coordinates.
WinOffsetRect	Offsets a rectangle.
WinPtInRect	Queries whether a point lies within a rectangle.
WinQueryUpdateRect	Returns the rectangle that bounds the update region of a specified window.
WinQueryWindowRect	Returns a window rectangle.
WinScrollWindow	Scrolls the contents of a window rectangle.
WinSetRect	Sets rectangle coordinates.
WinSetRectEmpty	Sets a rectangle empty.
WinShowTrackRect	Hides or shows the tracking rectangle.
WinSubtractRect	Subtracts one rectangle from another.
WinTrackRect	Draws a tracking rectangle.
WinUnionRect	Calculates a rectangle that bounds the two source rectangles.
WinValidateRect	Subtracts a rectangle from the update region of an asynchronous paint window, marking that part of the window as visually valid.

Table 29-2. Window-Drawing Structures	
Structure Name	**Description**
POINTL	Point structure (long integer).
RECTL	Rectangle structure.

Chapter 30. Hooks

A *hook* is a point in a system-defined function where an application can supply additional code that the system processes as though it were part of the function. This chapter describes how to use hooks in your PM applications.

About Hooks

Many operating system functions provide points where an application can *hook in* its own code to enhance or override the default processing of the function. OS/2 contains many types of hooks, and the system maintains a separate *hook list* for each type.

Hook Lists

A *hook list* contains the addresses of the functions that the system calls while processing a hook. An application can take advantage of a particular type of hook by defining a hook function and using the WinSetHook function to enter the address of the function in the corresponding hook list. The application uses one of the following constants in the WinSetHook function to specify the hook type:

Table 30-1. Hook Types

Type	**Description**
HK_CODEPAGECHANGED	Enables applications to determine when the code page changes.
HK_FINDWORD	Enables applications to control where the WinDrawText function places line breaks.
HK_HELP	Monitors the WM_HELP message.
HK_INPUT	Monitors messages in the specified message queue.
HK_JOURNALPLAYBACK	Enables applications to insert messages into the system message queue.
HK_JOURNALRECORD	Allows applications to record mouse and keyboard input messages.
HK_MSGFILTER	Monitors input events during system modal loops.
HK_SENDMSG	Monitors messages sent by using the WinSendMsg function.

While executing a function that contains a hook, the system checks for any function addresses in the hook list that correspond to the type of hook. If an address is found, the system tries to locate and execute the function.

Message-Monitoring Hooks

Most hooks enable an application to monitor some aspect of the message stream. For example, the input hook enables an application to monitor all messages posted to a particular message queue.

A hook function can be associated with the system message queue, so that it monitors messages for all applications, or with the message queue of an individual thread, so that it monitors messages for that thread only.

Hook functions associated with the system message queue can be called in the context of any application. You must define system-queue hook functions in separate dynamic link library (DLL) modules, because it is not possible to call application-module procedures from other applications.

Hook functions associated with the message queue of a thread are called only in the context of that thread. This kind of hook function typically is a locally-defined function.

The function addresses in the hook lists associated with most message-monitoring hooks are linked to form chains. The system passes a message to each hook function referenced in the list, one after the other. Each function can modify the message or stop its progress through the chain, preventing it from reaching the next hook or destination window. The system calls chained hook functions in last-installed, first-called order.

Hook Functions

Each type of hook passes a characteristic set of arguments to the functions referenced in the corresponding hook list. For an application to use a particular hook, it must define a function that processes those arguments and enter the address of the function in the hook list. This section describes the types of hooks available in OS/2 and the requirements of the functions that process each hook type.

Input Hook

The *input hook* enables an application to monitor the system message queue or an application message queue. The system calls an input-hook function whenever the WinGetMsg or WinPeekMsg function is about to return a message. Typically, an application uses the input hook to monitor mouse and keyboard input and other messages posted to a queue.

The syntax for an input-hook function is as follows:

```
BOOL EXPENTRY InputHook(HAB hab, PQMSG pQmsg, ULONG fs)
```

The *pQmsg* parameter is a far pointer to a QMSG structure that contains information about the message.

The *fs* parameter of the InputHook function can contain the following flags from the WinPeekMsg function, indicating whether or not the message is removed from the queue:

```
PM_NOREMOVE
PM_REMOVE
```

If an input-hook function returns TRUE, the system does not pass the message to the rest of the hook chain or to the application. If the function returns FALSE, the system passes the message to the next hook in the chain, or to the application if no other hooks exist.

An input-hook function can modify a message by changing the contents of the QMSG structure, then return FALSE to pass the modified message to the rest of the chain. The following problems can occur when a hook modifies a message:

- If the caller uses the WinPeekMsg or WinGetMsg function with a message filter range (msgFilterFirst through msgFilterLast), the message is checked before the hook functions are called, not after. If the input-hook function modifies the **msg** field of the QMSG structure, the caller can receive messages that are not in the range of the message filter of the caller.
- If the input-hook function changes a WM_CHAR message from one character into another—for example, if the function modifies all Tab messages into F6 messages—an application that depends on the key state is unable to interpret the result. (When the Tab key is translated into the F6 key, the application receives the F6 keystroke and enters a process loop, waiting for the F6 key to be released; the application calls the WinGetKeyState function with the HWND_DESKTOP and VK_F6 arguments).

Send-Message Hook

The *send-message hook* enables an application to monitor messages that the system does not post to a queue. The system calls a send-message hook function while processing the WinSendMsg function, before delivering the message to the recipient window. By installing an input-hook function and a send-message hook function, an application can monitor all window messages effectively.

The syntax for a send-message hook function is as follows:

```
VOID EXPENTRY SendMsgHook(HAB hab, PSMHSTRUCT psmh,
    BOOL fInterTask)
```

The *psmh* parameter is a far pointer to an SMHSTRUCT structure that contains information about the message.

The *fInterTask* parameter is TRUE if the message is sent between two threads, or FALSE if the message is sent within a thread.

A send-message hook function does not return a value, and the next function in the chain always is called. The function can modify values in the SMHSTRUCT structure before returning.

Message-Filter Hook

The *message-filter hook* allows an application to provide input filtering (such as monitoring hot keys) during system-modal loops. The system calls a message-filter hook function while tracking the window size and movement, displaying a modal dialog window or message box, tracking a scroll-bar, and during window-enumeration operations.

The syntax of a message-filter hook function is as follows:

```
BOOL EXPENTRY MsgFilterHook(HAB hab, PQMSG pQmsg, ULONG msgf)
```

The *msgf* parameter has the following three values:

Table 30-2. Message Filter Hook Parameter Values	
Value	**Meaning**
MSGF_DIALOGBOX	Message originated while processing a modal dialog window or a message box.
MSGF_MESSAGEBOX	Message originated while processing a message box.
MSGF_TRACK	Message originated while tracking a control (such as a scroll bar).

The *pQmsg* parameter of the MsgFilterHook function is a pointer to a QMSG structure containing information about the message.

If a message-filter hook function returns TRUE, the system does not pass the message to the rest of the hook chain or to the application. If the function returns FALSE, the system passes the message to the next hook function in the chain, or to the application if no other functions exist.

This hook enables applications to perform message filtering during modal loops that is equivalent to the typical filtering for the main message loop. For example, applications often examine a new message in the main event loop between the time they retrieve the message from the queue and the time they dispatch it, performing special processing as appropriate. An application usually cannot do this sort of filtering during a modal loop, since the system executes the loop created by the WinGetMsg and WinDispatchMsg functions. If an application installs a message-filter hook function, the system calls the function between WinGetMsg and WinDispatchMsg in the modal processing loop.

An application also can call the message-filter hook function directly by calling the WinCallMsgFilter function. With this function, the application can use the same code it uses in the main message loop to filter messages during modal loops. To do so, the application encapsulates the filtering operations in a message-filter hook function and calls WinCallMsgFilter between the calls to the WinGetMsg and WinDispatchMsg functions, as shown in the following code fragment:

```
while (WinGetMsg(hab, (PQMSG) &qmsg, (HWND) NULL, 0, 0)) {
    if (!WinCallMsgFilter(hab, (PQMSG) &qmsg, 0))
        WinDispatchMsg(hab, (PQMSG) &qmsg);
}
```

The last argument of the WinCallMsgFilter function is passed to the hook function; the application can enter any value. The hook function can use that value to determine from where the function was called by defining a constant such as MSGF_MAINLOOP.

Journal-Record Hook

The *journal-record hook* allows an application to monitor the system message queue and record input events. Typically, an application uses this hook to record a sequence of mouse and keyboard events that it can play back later by using the journal-playback hook. A journal-record hook function can be associated only with the system message queue.

The syntax for a journal-record hook function is as follows:

```
VOID EXPENTRY JournalRecordHook(HAB hab, PQMSG pQmsg)
```

The *pQmsg* parameter is a pointer to a QMSG structure containing information about the message. The system calls the journal-record hook function after processing the raw input enough to create valid WM_CHAR or mouse messages and after setting the **window-handle** field of the QMSG structure.

A journal-record hook function does not return a value, and the system always calls the next function in the chain. Typically, a journal-record hook function saves the input events to a disk file, to be played back later. The **hwnd** field of the QMSG structure is not important and is ignored when the message is read back.

The following messages are passed to the journal-record hook:

```
WM_CHAR
WM_BUTTON1DOWN
WM_BUTTON1UP
WM_BUTTON2DOWN
WM_BUTTON2UP
WM_BUTTON3DOWN
WM_BUTTON3UP
WM_MOUSEMOVE
```

The positions stored in the mouse messages are in screen coordinates. The system does not combine mouse clicks into double clicks before calling the hook, since there is no guarantee that both clicks will be in the same window when they are read back.

The system passes a WM_JOURNALNOTIFY message to the journal-record hook function whenever an application calls the WinGetPhysKeyState or WinQueryQueueStatus function. This message is necessary because the system message queue is only one message deep while a playback hook is active. For example, the user might press the A, B, and C keys while in record mode. While the application is processing the *A* character message, the B key might be down; WinGetPhysKeyState returns this information. However, during playback mode, the system knows only that it currently is processing the A key.

Journal-Playback Hook

The journal-playback hook enables an application to insert messages into the system message queue. Typically, an application uses this hook to play back a series of mouse and keyboard events that were recorded earlier using the journal-record hook. A journal-playback hook function can be associated only with the system message queue.

Regular mouse and keyboard input is disabled as long as a journal-playback hook is installed. It is important to notice that, since mouse and keyboard input are disabled, this hook easily can hang the system.

The syntax for a journal-playback hook function is as follows:

```
ULONG EXPENTRY JournalPlaybackHook(HAB hab, BOOL fSkip,
    PQMSG pQmsg)
```

The *pQmsg* parameter is a pointer to a QMSG structure that the journal-playback hook function fills in with the message to be played back. If the *fSkip* parameter is FALSE, the function should fill in the QMSG structure with the current recorded message. The function returns the same message each time it is called, until *fSkip* is TRUE. The same message is returned many times if an application is examining the queue but not removing the message. If *fSkip* is TRUE, the function should advance to the next message without filling in the QMSG structure, since the *pQmsg* parameter is NULL when *fSkip* is TRUE.

The journal-playback hook returns a ULONG time-out value that tells the system how many milliseconds to wait before processing the current message from the playback hook. This enables the hook to control the timing of the events it plays back.

The **time** field of the QMSG structure is filled in with the current time before the playback hook is called. The hook should use the time stored in this field, instead of the system clock to set up delays between events.

Help Hook

The *help hook* allows an application to include online help. The system calls a help-hook function during the default processing of the WM_HELP message. Help processing is done in two stages: creating the WM_HELP message and calling the help hook. The WM_HELP message can come from the following sources:

- From a WM_CHAR message, after translation by an ACCEL structure with the AF_HELP style. The default system accelerator table translates the F1 key into a help message. The WM_HELP message is posted to the current focus window, which can be a menu, a button, a frame, or your client window.
- From a menu-bar selection, when the MIS_HELP style is specified for the menu-bar item. The WM_HELP message is posted to the current focus window.
- From a dialog-window pushbutton, when the BS_HELP style is specified for the pushbutton. The WM_HELP message is posted to the owner window of the button, which normally is the dialog window.
- From a message box, when the MB_HELP style is specified for the message box. The WM_HELP message is posted to the message box.

The WM_HELP message is posted to the current focus window. The default processing in the WinDefWindowProc function is to pass the message up to the parent window. If the message reaches the client window, it can be processed there. If the message reaches a frame window, the default frame-window procedure calls the help hook. The help hook also is called if a WM_HELP message is generated while the application is in menu mode—that is, while a selection is being made from a menu.

The syntax for a help-hook function is as follows:

```
BOOL EXPENTRY HelpHook(HAB hab, ULONG usMode, ULONG idTopic,
    ULONG idSubTopic, PRECTL prcPosition)
```

If a help-hook function returns TRUE, the system does not call the next help-hook function in the chain. If the function returns FALSE, the system calls the next help-hook function in the chain. The arguments passed to the function provide contextual information, such as the screen coordinates of the focus window and whether the message originated in a message box or a menu.

The WM_HELP message often goes to a frame window instead of to the client window. The frame window processes a WM_HELP message as follows:

- If the window with the focus is the FID_CLIENT window, the frame window passes the WM_HELP message to the FID_CLIENT window.
- If the parent of the window with the focus is the FID_CLIENT frame-control window, the frame window calls the help hook, specifying the following:

```
Mode = HLPM_FRAME
Topic = frame-window identifier
Subtopic = focus-window identifier
Position = screen coordinates of focus window
```

- If the parent of the focus window is not an FID_CLIENT window (it could be the frame window or a second-level dialog window), the frame window calls the help hook, specifying the following:

```
Mode = HLPM_WINDOW
Topic = identifier of parent of focus window
Subtopic = focus-window identifier
Position = screen coordinates of focus window
```

An application receives the WM_HELP message in its dialog-window procedure. The application can ignore the message, in which case the frame-window action occurs as described, or the application can handle the WM_HELP message directly.

Menu windows receive a WM_HELP message when the user presses the Help accelerator key (F1 by default) while a menu is displayed. Menu windows process WM_HELP messages by calling the help hook, specifying the following:

```
Mode = HLPM_MENU
Topic = identifier of pull-down menu
Subtopic = identifier of selected item in pull-down menu
Position = screen coordinates of selected item
```

A help-hook function should respond by displaying information about the selected menu item.

The WinDefWindowProc function processes WM_HELP messages by passing the message to the parent window. Typically, the message moves up the parent chain until it arrives at a frame window.

Find-Word Hook

The *find-word hook* allows an application to control where the WinDrawText function breaks a character string that is too wide for the drawing rectangle. The system calls this hook from within the WinDrawText function, if the DT_WORDBREAK flag is set. Typically, this hook is used in applications that use double-byte character sets to avoid awkward line breaks.

The syntax for a find-word hook function is as follows:

```
ULONG EXPENTRY FindWordHook(ULONG ulCodePage, PSZ pszText, ULONG cb,
                            ULONG ich, PULONG pichStart, PULONG pichEnd,
                            PULONG pichNext)
```

The *usCodePage* parameter contains the code page identifier of the string to be formatted; the *pszText* parameter contains a pointer to the actual string.

The *cb* parameter contains a value specifying the number of bytes in the string. This value is 0 if the string is null terminated.

The *ich* parameter contains the index of the character in the string that intersects the right edge of the drawing rectangle.

A find-word hook function uses these four parameters to determine the word that contains the intersecting character. It then fills the remaining three parameters, *pichStart*, *pichEnd*, and *pichNext*, with the indexes of the starting character of the word, ending character of the word, and starting character of the next word in the string.

If the find-word hook function returns TRUE, WinDrawText draws the string only up to, but not including, the specified word. If the function returns FALSE, WinDrawText formats the string in the default manner.

Codepage-Changed Hook

The *codepage-changed hook* notifies an application when the code page associated with the specified message queue has been changed. The system calls a codepage-changed hook function after setting the new code page. Typically, the codepage-changed hook is used in applications that support multiple languages.

The syntax for a codepage-changed hook function is as follows:

```
VOID EXPENTRY CodePageChangedHook(HMQ hmq, ULONG ulOldCodepage,
    ULONG  ulNewCodepage)
```

The *hmq* parameter receives the handle of the message queue that is changing its codepage. The usOldCodepage is the codepage identifier of the previous code page; usNewCodepage is the identifier of the new code page.

A codepage-changed hook function does not return a value, and the system always calls the next function in the chain.

Using Hooks

This section explains how to perform the following tasks:

- Install and release hook functions.
- Monitor messages in a message queue.
- Monitor messages sent between windows.
- Record and play back input events.
- Specify line breaks for the WinDrawText function.

Installing and Releasing Hook Functions

You can install hook functions by calling WinSetHook, specifying the type of hook that calls the function—whether the function is to be associated with the system message queue or with the queue of a particular thread—and a pointer to a function entry point. The following code fragment shows how to install a hook function in the message queue of a thread:

```
BOOL EXPENTRY MyInputHook(HAB, PQMSG, USHORT);
HAB hab;
HMQ hmq;

WinSetHook(hab,          /* Anchor block handle           */
    hmq,                 /* Thread message queue          */
    HK_INPUT,            /* Called by the input hook      */
    (PFN) MyInputHook,   /* Address of input-hook function */
    (HMODULE)NULL);      /* Function is in appl. module   */
```

Place hook functions associated with the system message queue in a dynamic link library (DLL) separate from the application that installs the hook function. The installing application needs the handle of the DLL module before it can install the hook function. The DosLoadModule function, given the name of the DLL, returns the handle of the DLL module. Once you have the handle, you can call DosQueryProcAddr to obtain the address of the hook function. Finally, use the WinSetHook function to install the hook function address in the appropriate hook list. WinSetHook passes the module handle, a pointer to the hook-function entry point, and NULL for the message-queue argument, indicating that the hook function should be associated with the system queue.

You can release a hook function (that is, remove its address from the hook list) by calling the WinReleaseHook function with the same arguments that you used when installing the hook function. Release all hook functions before the application terminates, even though the system automatically releases them if the application does not.

A system hook can be released by using the WinReleaseHook function, but the DLL module containing the hook function is not freed because system-hook functions are called in the process context of every PM application in the system, causing an implicit call to DosLoadModule for all those processes. Since a call to the DosFreeModule function cannot be made for another process, there is no way to free the DLL modules. (However, since the hook function is no longer called, the DLL segments of the module may be discarded or swapped.

An alternative method for installing a system-queue hook function is to provide an installation function in the DLL along with the hook function. With this method, the installing application does not need the handle of the DLL module. By linking with the DLL, the application gains access to the installation function. The installation function can supply the DLL module handle and other details in the call to the WinSetHook function. The DLL also can contain a function that releases the system-queue hook function. The application can call this hook-releasing function when it terminates.

Summary

Following are the OS/2 functions and structures used with hook controls.

Table 30-3. Hook Functions

Function name	Description
WinCallMsgFilter	Calls a message-filter hook.
WinReleaseHook	Releases an application hook from a hook chain.
WinSetHook	Installs an application procedure in a specified hook chain.

Table 30-4. Hook Functions

Structure name	Description
QMSG	Message structure.
SMHSTRUCT	Send-message-hook structure.

Chapter 31. Clipboards

The *clipboard* is a small amount of system-managed random-access memory (RAM) for *user-driven* data exchange. This is in contrast with dynamic data exchange (DDE), which is application driven. While the clipboard only stores *pointers* to data, its associated set of functions can be used in applications to move and exchange data. This chapter describes how to use the clipboard in PM applications.

About the Clipboard

The clipboard enables the user to move data in a single application or exchange data among applications. Typically, a user selects data in the application using the mouse or keyboard, then initiates a cut, copy, or paste operation on that selection. Figure 31-1 is an example of copying data from one application, and Figure 31-2 illustrates pasting that same data in another application by way of the clipboard.

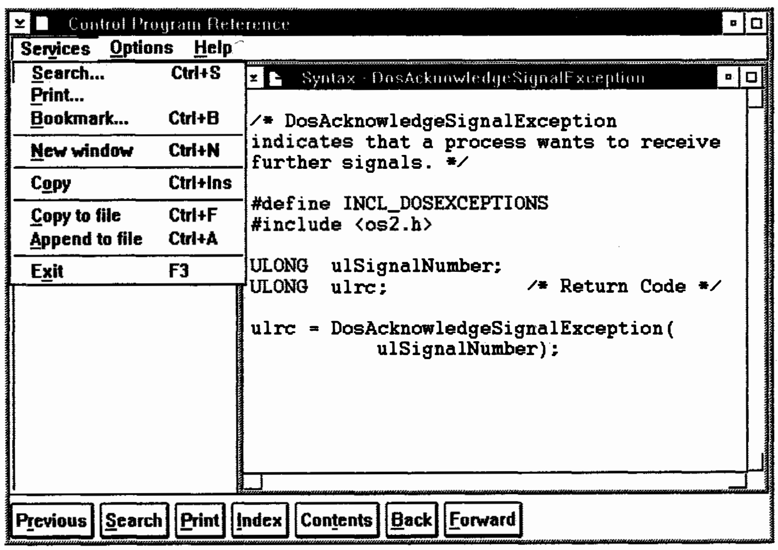


Figure 31-1. A Copy Operation Between Applications Using the Clipboard

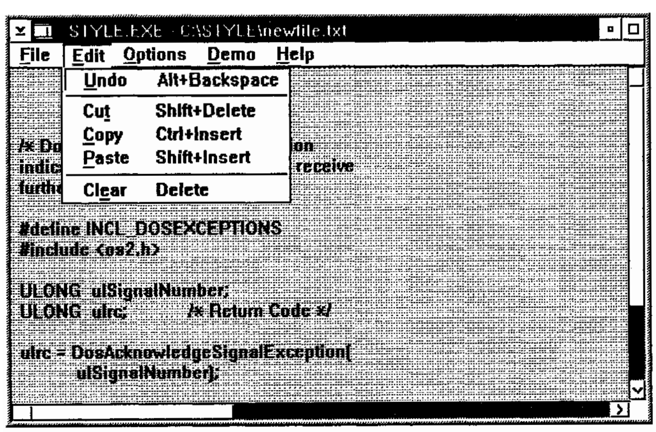


Figure 31-2. A Paste Operation Between Applications Using the Clipboard

Descriptions of these operations are in the following table:

Table 31-1. Operations on Clipboard Data	
Operation	**Description**
Cut	Deletes the selected data from the application and copies it to the clipboard. Any previous contents of the clipboard are destroyed.
Copy	Copies the selected data to the clipboard. The selection remains unchanged. Previous contents of the clipboard are destroyed.
Paste	Deletes the selected data from the application and replaces it with the contents of the clipboard. The contents of the clipboard are not changed.

An application should not perform any clipboard operations unless the user initiates them explicitly. Other OS/2 features, such as pipes, queues, shared memory, and especially DDE should be used when data exchange is needed without user involvement. For example, an application that continuously passes remotely collected data to a data-analysis application must not use the clipboard. Such an application, instead, should use the other interprocess data-communication capabilities of the operating system.

The data on the clipboard is maintained in memory only. Clipboard data is lost when the computer is turned off.

Shared Memory and the Clipboard

An application must store, in shared memory, text data that is destined for the clipboard. To do so, the application calls the DosAllocSharedMem function with the OBJ_GIVEABLE attribute to allocate a shared memory object, and then copies the text data to the object. The application passes the clipboard a pointer, which the clipboard uses to access the shared memory object. Clipboard functions use the CFI_POINTER flag to indicate text data stored in a shared memory object.

To pass a bit map or metafile to the clipboard, an application passes the clipboard a bit map or metafile handle. The clipboard functions make the bit map or metafile *shareable*. The CFI_HANDLE flag is used in clipboard functions to indicate bit map or metafile data.

After closing the clipboard, an application no longer can access the data it passed to the clipboard. Likewise, when an application requests data from the clipboard, it receives a pointer or handle that is good only until the application closes the clipboard. Typically, the application either uses the data immediately before closing the clipboard, or it copies the data to local memory for future use, then closes the clipboard.

Clipboard Operations

An application uses the clipboard when cutting, copying, or pasting data. Typically, an application places data on the clipboard for cut and copy operations and removes data from the clipboard for paste operations. The following paragraphs describe all these clipboard operations.

Cut and Copy Operations

To put data on the clipboard, an application first calls the WinOpenClipbrd function to verify that other applications are not trying to retrieve or set clipboard data. The WinOpenClipbrd function does not return if another thread has the clipboard open; it waits until either the clipboard is free or there is a message in the message queue of the calling thread. In practice, the WinOpenClipbrd function waits until the clipboard is available or until the calling application responds to a message. If the clipboard cannot be opened before a message arrives, the application receives the message, and the WinOpenClipbrd function continues to try to open the clipboard. The WinOpenClipbrd function does not return until the clipboard is open. However, the application continues to receive messages.

Once an application successfully opens the clipboard, it must remove any previously stored data on the clipboard by calling the WinEmptyClipbrd function. If the clipboard is not cleared, writing an existing format on the clipboard replaces the old data in that format with the new data. Old data in other formats remains on the clipboard.

After emptying the clipboard, an application should write its data to the clipboard in as many standard formats as possible. For each format, the application passes the data to the clipboard by calling the WinSetClipbrdData function, specifying each data format. The clipboard is not cleared when a new format is written to it; all new data formats coexist on the clipboard until it is cleared by the next clipboard user.

If an application passes NULL as the *ulData* parameter of the WinSetClipbrdData function, applications must render the data on request.

Finally, when an application finishes writing the clipboard data, it must release the clipboard by calling the WinCloseClipbrd function so that other applications can use the clipboard.

Paste Operation

To retrieve data from the clipboard, an application first must call the WinOpenClipbrd function to verify that no other applications are trying to retrieve or set the clipboard data.

Once an application successfully opens the clipboard, it calls the WinQueryClipbrdData function, specifying a preferred format. If that format is not available (indicated by a NULL return from the WinQueryClipbrdData function) the application should continue to call WinQueryClipbrdData for other possible formats until it either receives the data or runs out of format choices.

If the clipboard contains one of the requested formats, the WinQueryClipbrdData function returns a 32-bit integer, the meaning of which depends on the particular format. For text data, the return value is a pointer to a shareable memory object containing the text. For bit map data, the return value is a bit map handle. For metafile data, the return value is a metafile handle.

It is important that an application use the WinCloseClipbrd function to close the clipboard as soon as possible so that other applications can access it.

Standard Clipboard-Data Formats

The clipboard can accept data in three standard formats: text, bit map, and metafile. Applications can either use these formats or create their own private formats.

All PM applications can access the clipboard, so applications can copy to the clipboard the same selection of data in many different formats. For example, a word processor that supports multiple fonts might write the same selection of text to the clipboard in three different formats: straight text, *rich* text, and metafile. Then, another application (pasting from the clipboard) could choose the appropriate format.

Applications can use the following constants to specify the standard clipboard-data formats:

Table 31-2. Clipboard Data Formats

Format	Description
CF_BITMAP	Specifies that the data in the clipboard is a bit map.
CF_DSPBITMAP	Specifies that the data in the clipboard is a bit map representation of a private-data format. The clipboard viewer uses this format to display a private format.
CF_DSPMETAFILE	Specifies that the data in the clipboard is a metafile representation of a private-data format. The clipboard viewer uses this format to display a private format.
CF_DSPTEXT	Specifies that the data in the clipboard is a text representation of a private-data format. The clipboard viewer uses this format to display a private format.
CF_METAFILE	Specifies that the data in the clipboard is a metafile.
CF_TEXT	Specifies that the data in the clipboard is an array of text characters. These characters can include *newline* characters to indicate line breaks. The NULL character indicates the end of the text data.

Private Clipboard-Data Formats

Applications that use the clipboard to move data within the documents of the application can use private clipboard-data formats when standard formats are insufficient for representing clipboard data. For example, a word processor might have a rich-text format that contains font and style information in addition to the usual text characters. Clearly, if the word processor uses the clipboard to support cut, copy, and paste operations for moving data in its documents, a standard text format will be inadequate.

In such case, the word processor should write at least two formats to the clipboard for each cut or copy operation: a standard text format representing the text of the current selection and a private rich-text format representing the true state of the selection. If the word processor performs a paste operation by using clipboard data, it can use the rich-text format to retain all formatting. If another application requests the same data, it can use the standard-text format if it does not recognize the private format. Also, the word processor should be able to render data in CF_BITMAP and CF_METAFILE formats for painting and drawing applications.

Format Identification Number

Each private format must have a unique identification number. To obtain an identification number, the application registers the name of the private format in the system atom table. The system assigns a unique identification number for the format name. Other applications having access to the format name can query the system atom table for the format identification number.

An application can interpret its own private formats and request them from the clipboard for cutting and pasting its own data. Other applications that know the private-format identification number also can interpret the formatted data.

Display Formats

OS/2 provides three standard display formats for applications that use private formats: CF_DSPTEXT, CF_DSPBITMAP, and CF_DSPMETAFILE. These formats correspond to the standard text, bit map, and metafile formats, with the exception that they are intended for use only by the clipboard viewer. An application that uses a private format should write one of the DSP formats that approximates the appearance of the private data so that the clipboard viewer can display the data regardless of the format. For example, a word processor using the rich-text format also would write a CF_DSPBITMAP formatted picture of the selected text that contains all the type fonts and styles.

Notice that you can choose delayed rendering for DSP formats because there might not always be a clipboard viewer active on the screen. With delayed rendering, an application actually does not render the format unless it is requested to do so.

Delayed Rendering

An application can pass NULL as the *ulData* parameter of the WinSetClipbrdData function instead of a pointer or a handle. This indicates that the data is rendered only when another application requests it from the clipboard. This is useful if an application supports several clipboard formats that are time-consuming to render. With delayed rendering, an application can send NULL handles for each clipboard format that it supports and render individual formats only when the format actually is requested from the clipboard. An application can either write data for standard formats or choose delayed rendering for more complex formats.

When an application uses delayed rendering for one or more of its clipboard formats, it must become the clipboard owner. As long as the application is the clipboard owner, it receives a WM_RENDERFMT message whenever a request is received by the clipboard for a format using delayed rendering. When the application receives such a message, it renders the data and passes the pointer or handle to the clipboard by calling the WinSetClipbrdData function. The rules for shared-memory access for rendered data are the same as those for standard clipboard data. This simply is a delayed execution of the operation that occurs if the data does not have delayed rendering.

The clipboard owner, with one or more delayed-rendering formats on the clipboard, receives a WM_RENDERALLFMTS message just before the clipboard-owner application terminates. This ensures that the application renders all of its data before terminating.

Clipboard Viewer

A window can become a clipboard viewer and display the current contents of the clipboard. The clipboard viewer is informed whenever the clipboard contents change. Typically, the clipboard viewer is a window that can draw the standard clipboard formats. The clipboard viewer is a convenience for the user; it does not have any effect on the data-transaction functions of the clipboard.

To create a clipboard viewer, an application calls WinSetClipbrdViewer, specifying the window in which the clipboard data will be displayed. Usually this is the client window of an application. There can be only one clipboard viewer at any time in the system, so setting a clipboard viewer replaces any previous clipboard viewer. The WinQueryClipbrdViewer function receives the handle to the current clipboard viewer so that the application can reset it when finished with the clipboard viewer.

Once a window becomes the clipboard viewer, it receives WM_DRAWCLIPBOARD messages whenever the contents of the clipboard change. The window should respond to these messages by drawing the contents of the clipboard.

The clipboard viewer displays all the standard formats and should process CFI_OWNERDISPLAY items by sending the appropriate message to the clipboard owner.

The clipboard viewer cannot display private-format data. For this reason, an application that writes private-format data to the clipboard also must write the data in one of the three standard-display formats: CF_DSPTEXT, CF_DSPBITMAP, or CF_DSPMETAFILE.

If a standard format is not provided in addition to the private formats, the clipboard owner must draw the clipboard data in the clipboard-viewer window. An application uses the CFI_OWNERDRAW flag to identify clipboard data that the clipboard owner draws. When the clipboard viewer encounters data with the CFI_OWNERDRAW flag set, it sends WM_PAINTCLIPBOARD messages to the clipboard owner whenever the data must be drawn, scrolled, or sized.

The clipboard viewer determines the attributes of a particular clipboard format by calling the WinQueryClipbrdFmtInfo function. The identity of the current owner is found by calling the WinQueryClipbrdOwner function.

Clipboard Owner

The *clipboard owner* is any application window connected to the clipboard data. Following are situations in which an application would call WinSetClipbrdOwner to become the clipboard owner:

- The application calling WinSetClipbrdData passes a NULL pointer or handle to the clipboard, indicating that the application renders the data in a particular format on request. As a result, the system sends rendering requests to the current clipboard owner.
- The application calling WinSetClipbrdData passes data with the CFI_OWNERFREE attribute, indicating that the application frees memory for data when the clipboard is emptied. As a result, the system sends owner-free requests to the current clipboard owner.
- The application calling WinSetClipbrdData passes data with the CFI_OWNERDISPLAY attribute, indicating that the owner application draws the data in the clipboard viewer. As a result, the clipboard viewer sends drawing-related requests to the current clipboard owner.

The window specified in the call to the WinSetClipbrdOwner function responds to the following messages:

Table 31-3. Messages Handled by Clipboard Owner	
Message	**Description**
WM_RENDERFMT	Sent by the system to the clipboard owner when a particular format with delayed rendering must be rendered. The receiver must render the data in the specified format and pass it to the clipboard by calling the WinSetClipbrdData function.
WM_RENDERALLFMTS	Sent by the system to the clipboard owner just before the owner application terminates. The receiver must render the clipboard data in all formats on the clipboard with delayed rendering. It must pass the data for each format to the clipboard by calling the WinSetClipbrdData function.
WM_DESTROYCLIPBOARD	Sent by the system to the clipboard owner when the clipboard is cleared by another application calling the WinEmptyClipbrd function. The receiver must free the memory occupied by any clipboard formats using the CFI_OWNERFREE attribute.
WM_SIZECLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains the data handle with the CFI_OWNERDISPLAY attribute and when the clipboard-viewer changes size. When the clipboard viewer is being destroyed or reduced to an icon, this message is sent with the coordinates of the opposite corners set to (0,0), which permits the owner to free its display resources.
WM_VSCROLLCLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the CFI_OWNERDISPLAY attribute and when an event occurs in the clipboard-viewer scroll bars. The receiver must respond to this message by scrolling the image, invalidating the appropriate area of the clipboard viewer, and updating the slider position.
WM_HSCROLLCLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the CFI_OWNERDISPLAY attribute and when an event occurs in the scroll bars of the clipboard viewer. The receiver must respond to this message by scrolling the image, invalidating the appropriate area of the clipboard viewer, and updating the slider position.
WM_PAINTCLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the CFI_OWNERDISPLAY attribute and when the clipboard-viewer client area needs repainting. The receiver must respond to this message by painting the requested format (by calling WinGetPS for the window handle of the clipboard viewer).

An application automatically loses ownership of the clipboard when the clipboard data is cleared by the WinEmptyClipbrd function. Ownership is necessary only when data is present on the clipboard. Typically, an application loses ownership when another application places data on the clipboard.

Using the Clipboard

You can use the clipboard functions to perform the following tasks:

- Put data on the clipboard.
- Retrieve data from the clipboard.
- View data on the clipboard.

Putting Data on the Clipboard

The following code fragment shows how an application places text data on the clipboard, how it opens the clipboard, copies the text to a shared memory object, empties the clipboard, and passes the pointer to the clipboard:

```
#define MAXSTR   1024

PSZ  pszSrc, pszDest;
BOOL fSuccess;
CHAR szClipString[MAXSTR];
HAB  hab;

    .
    . /* Get character string (szClipString). */
    .

if (WinOpenClipbrd(hab)) {

    /* Allocate a shared memory object for the text data. */
    if (!(fSuccess = DosAllocSharedMem(
            (PVOID)&pszDest,       /* Pointer to shared memory object */
            NULL,                  /* Use unnamed shared memory       */
            strlen(szClipString)+1,/* Amount of memory to allocate    */
            PAG_WRITE  |           /* Allow write access              */
            PAG_COMMIT |           /* Commit the shared memory        */
            OBJ_GIVEABLE))) {      /* Make pointer giveable           */

        /* Set up the source pointer to point to text. */
        pszSrc = szClipString;

        /* Copy the string to the allocated memory. */
        while (*pszDest++ = *pszSrc++);

        /* Clear old data from the clipboard. */
        WinEmptyClipbrd(hab);

        /*
         * Pass the pointer to the clipboard in CF_TEXT format. Notice
         * that the pointer must be a ULONG value.
         */

        fSuccess = WinSetClipbrdData(hab, /* Anchor-block handle    */
            (ULONG) pszDest,              /* Pointer to text data   */
            CF_TEXT,                      /* Data is in text format */
            CFI_POINTER);                 /* Passing a pointer      */

        /* Close the clipboard. */
        WinCloseClipbrd(hab);
    }
}
```

Retrieving Data from the Clipboard

The following code fragment shows how to open the clipboard, retrieve data in the requested format, copy the data to local memory, and close the clipboard:

```
PSZ pszClipText, pszLocalText;

if (WinOpenClipbrd(hab)) {
    if (pszClipText = (PSZ) WinQueryClipbrdData(hab, CF_TEXT)) {

        /* Copy text from the shared memory object to local memory. */
        while (*pszLocalText++ = *pszClipText++);
    }
    WinCloseClipbrd(hab);
}
```

Viewing Data on the Clipboard

The following code fragment shows how a sample clipboard viewer responds to the WM_DRAWCLIPBOARD message, drawing text and bit map data in its window. Notice that the code uses the data retrieved from the clipboard before closing the clipboard. An alternative strategy is to copy the data and then close the clipboard. In any case, the original data from the clipboard cannot be used after the clipboard is closed.

```
PSZ     pszText;
HPS     hps;
RECTL   rcl;
HBITMAP hBitmap;
POINTL  ptlDest;

case WM_DRAWCLIPBOARD:
    if (!WinOpenClipbrd(hab))
        return 0;

    hps = WinGetPS(hwnd);  /* Get a presentation space for drawing */
    WinQueryWindowRect(hwnd, &rcl);/* Get dimensions of the window */

    if (pszText =(PSZ)WinQueryClipbrdData(hab, CF_TEXT)) {
        WinDrawText(hps,
            -1,                         /* Null-terminated string  */
            pszText,                    /* The string              */
            &rcl,                       /* Where to put the string */
            CLR_BLACK,                  /* Foreground color        */
            CLR_WHITE,                  /* Background color        */
            DT_CENTER | DT_VCENTER | DT_ERASERECT);
    }
    else if (hBitmap = (HBITMAP)WinQueryClipbrdData(hab, CF_BITMAP)) {
        ptlDest.x = ptlDest.y = 0;
        WinFillRect(hps, &rcl, CLR_WHITE);
        WinDrawBitmap(hps,
            hBitmap,
            NULL,                       /* Draws entire bit map    */
            &ptlDest,                   /* Destination             */
            CLR_BLACK,                  /* Foreground color        */
            CLR_WHITE,                  /* Background color        */
            DBM_NORMAL);                /* Bit map flags           */
    }

    /* Remove rectangle from the update region */
    WinValidateRect(hwnd, &rcl, FALSE);
    WinReleasePS(hps);          /* Release the presentation space.*/
    WinCloseClipbrd(hab);       /* Close the clipboard.           */
    return 0;
```

Summary

Following are the OS/2 functions and messages used with the clipboard:

Table 31-4. Clipboard Functions

Function name	Description
WinCloseClipbrd	Closes the clipboard, enabling other applications to open it by calling WinOpenClipbrd.
WinEmptyClipbrd	Empties the clipboard, removing and freeing all handles to data that is in the clipboard.
WinEnumClipbrdFmts	Enumerates the list of clipboard data formats available in the clipboard.
WinOpenClipbrd	Opens the clipboard.
WinQueryClipbrdData	Obtains a handle to the current clipboard data with a specified format.
WinQueryClipbrdFmtInfo	Determines whether a particular format of data is present in the clipboard; and, if so, provides information about that format.
WinQueryClipbrdOwner	Obtains any current clipboard owner window.
WinQueryClipbrdViewer	Obtains any current clipboard viewer window.
WinSetClipbrdData	Puts data into the clipboard.
WinSetClipbrdOwner	Sets the current clipboard owner window.
WinSetClipbrdViewer	Sets the current clipboard viewer window to a specified window.

Table 31-5. Clipboard Messages

Message	Description
WM_DESTROYCLIPBOARD	Sent to the clipboard owner when the clipboard is emptied through a call to WinEmptyClipbrd.
WM_DRAWCLIPBOARD	Sent to the clipboard viewer window whenever the contents of the clipboard change, that is, as a result of the WinCloseClipbrd call following a call to WinSetClipbrdData.
WM_HSCROLLCLIPBOARD	Sent to the clipboard owner window when the clipboard contains a data handle for the CFI_OWNERDISPLAY format.
WM_PAINTCLIPBOARD	Sent when the clipboard contains a data handle with the CFI_OWNERDISPLAY information flag set.
WM_RENDERALLFMTS	Sent to the application that owns the clipboard while the application is being destroyed.
WM_RENDERFMT	A request to the clipboard owner to render the data of the format specified in *usfmt.*
WM_SIZECLIPBOARD	Sent when the clipboard contains a data handle for the CFI_OWNERDISPLAY format, and the clipboard application window has changed size.
WM_VSCROLLCLIPBOARD	Sent to the clipboard owner window when the clipboard contains a data handle for the CFI_OWNERDISPLAY format.

Chapter 32. Dynamic Data Exchange

The Dynamic Data Exchange (DDE) protocol uses messages to communicate between applications that share data, and uses shared memory as the means of exchanging data between applications. Applications can use DDE for one-time data transfers and for ongoing exchanges in which the applications send updates to one another as new data becomes available. This chapter explains how to use DDE in PM applications.

About Dynamic Data Exchange

DDE is different from the clipboard data-transfer component that also is part of this operating system. One difference is that, almost always, the clipboard is used as a one-time response to a specific action by the user, such as choosing **Paste** from a menu. DDE, on the other hand, is often initiated by a user, but typically continues without the user's further involvement.

Client and Server Interaction

DDE transactions always take place between a *client* application and a *server* application. The client initiates the exchange by requesting data from the server. The server responds by providing the requested data to the client. A server can have many clients simultaneously; and a client can request data from multiple servers.

An application can be both a client and a server at the same time. For example, one application could receive data from another application as a client, and then act as a server by passing the data to yet another application. The important distinction between a client and a server is that *only the client initiates DDE transactions.*

A DDE transaction actually takes place between two windows, one for each of the participating applications. Applications open a window for each conversation in which they engage. (Notice that these windows are not necessarily visible.) A window is identified by its handle. The window belonging to the server application is the *server window*; the window belonging to the client application is the *client window*. Figure 32-1 illustrates how a link is established.

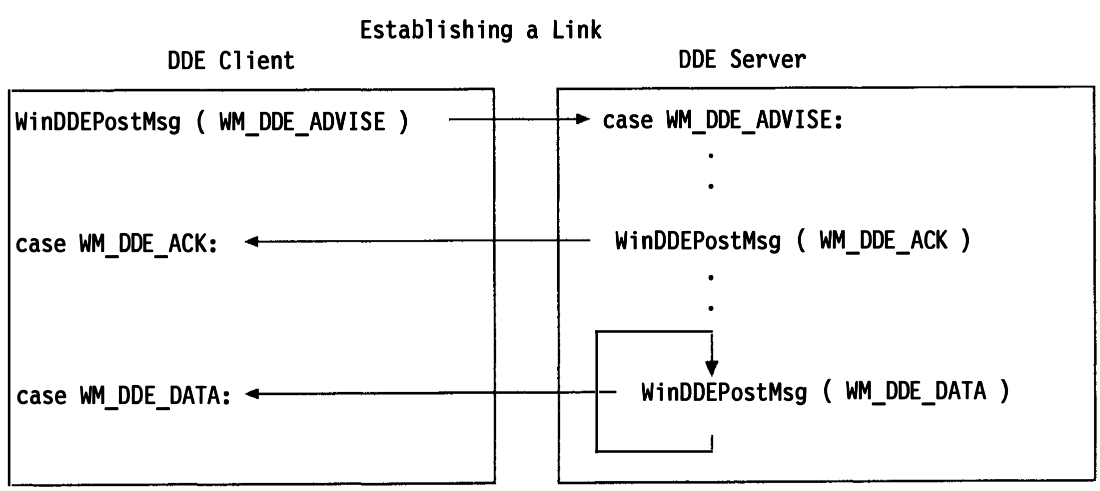


Figure 32-1. Linking a DDE Client with a DDE Server

Sample DDE System

There are many potential uses of DDE in real-time data acquisition applications. This section presents an example of one such use: a DDE-based, real-time system for tracking portfolios. Two hypothetical PM applications cooperate in this example. One application, named the *collector*, is a specialized interface that draws data from an online data service. The other application is a spreadsheet. Both applications use the DDE protocol.

The sample spreadsheet has the following layout:

	A	B	C	D
1	Stock	Shares	Price	Extension
2	BTRX	1000	148	148000
3	HLOW	2000	26	52000
4	WRLD	200	24	4800
5	ZMXI	2000	93	186000
6				390800

Without DDE, this spreadsheet could be updated by using the clipboard to copy numbers, manually, from the screen display of the collector application to the spreadsheet. This would require screen sharing or switching between applications, and also would require that the user pay close attention to the price data, and undertake the data exchange personally.

With DDE, this system could be much more automatic, providing the spreadsheet with the current values for multiple data items, without intervention by the user. DDE would enable the user to set up an exchange between the two applications that would keep the spreadsheet up to date whenever a change occurred in the value of specified stocks. Once this connection was established, the cell values in the spreadsheet always would reflect the most current data available from the collector. This system would facilitate the timely analysis of realtime data.

The usefulness of the DDE protocol is not restricted to specialized realtime data-acquisition applications. Productivity software, in general, can benefit significantly from the protocol. For example, suppose a monthly report is prepared using a graphics-and-text word processor, and that the report includes graphs generated in a separate business-graphics package. Without DDE, someone must manually copy and paste each month's new graphs into each month's report. With DDE, the word processor could establish a permanent link to the charting application so that any changes made to the charting document would be reflected in the word-processing document, either automatically or by request.

Detailed DDE Example

For a detailed view of the workings of the DDE protocol, here is an example that describes the collector and spreadsheet interaction and illustrates the forwarding of stock quotes from the collector application to the spreadsheet. For simplicity, this example is limited to the exchange of quotes for a single stock, BTRX.

The collector DDE server application is started first. Typically, applications designed to operate as dedicated DDE servers have a user interface for initialization, then run as icons at the bottom of the Presentation Manager screen. As part of the initialization process, the collector DDE server application performs the necessary tasks (such as entering passwords and testing) to ensure that it can provide data to clients.

The spreadsheet is started next, and the stock-portfolio document is loaded. At this time, the spreadsheet calls the WinDdeInitiate function, which sends a WM_DDE_INITIATE message to all top-level frame windows, that is, frame windows that have HWND_DESKTOP as their parent. The WM_DDE_INITIATE message is a request to initiate an exchange with an application on a specified topic—in this case, NYSE. An application can accept this message by responding with a positive WM_DDE_INITIATEACK message, or can decline it by passing the message on to the WinDefWindowProc function. If no application accepts the request, the spreadsheet assigns an error value to the external reference and its DDE activity concludes.

If the collector application acknowledges the request, the spreadsheet can use the newly established exchange to request the collector application to provide continuous updates on a specified data item. To make this request, the spreadsheet posts a WM_DDE_ADVISE message to the collector application (actually, to a window within the collector application that is acting as the message recipient for DDE messages), indicating that updates must be sent every time there is a new value available for the data item, BTRX, and that the updates should be in a particular format—for example, DDEFMT_TEXT.

Upon receiving this message, the collector application records the request in its database and posts a WM_DDE_ACK message to the spreadsheet. From then on, the collector application posts a WM_DDE_DATA message to the window in the spreadsheet that initiated the exchange whenever it receives a new BTRX stock quote from the server. Each of these messages carries a pointer to a shared-memory object that contains the data, rendered in the requested format. Whenever the spreadsheet receives such a message, it retrieves the data from the referenced memory object and uses the data to update the value of the cell containing the external reference.

The periodic updates continue until the spreadsheet document is closed. At that point, the spreadsheet application posts a WM_DDE_UNADVISE message to the collector application, indicating that further updating is unnecessary. Upon receipt of this message, the collector application removes the corresponding data request from its database and posts a positive WM_DDE_ACK message back to the spreadsheet.

Finally, unless the spreadsheet initiates other data exchanges under this same topic, it posts a WM_DDE_TERMINATE message to the collector application, indicating the end of the DDE transaction. The collector application responds with a WM_DDE_TERMINATE message.

Applications, Topics, and Items

DDE uses the three-level hierarchy—*application, topic,* and *item*— to uniquely identify a unit of data. An application is the name of the server from which the data is desired. A *topic* is a logical data context. For applications that operate on file-based documents, topics usually are file names; for other applications, they are other application-specific strings. An *item* is a data object that can be passed in a DDE transaction. For example, an item might be a single integer, a string, several

paragraphs of text, or a bit map. Using the collector and spreadsheet model described earlier, the application name is *collector*, the topic name is *NYSE*, and the item name is *BTRX*.

The System Topic

The *system topic* provides a context for information that may be of general interest to any partners in a DDE transaction. Applications are encouraged to support the system topic at all times. The string used for the system topic is defined in the PM header files as SZDDESYS_TOPIC.

DDE applications should request an exchange on the system topic with a NULL application name when they start up, to find out what kinds of information other DDE-capable programs can provide.

The system topic must support the following items as well as any other items the application uses:

Table 32-1. DDE System Topics

Item	Description
SZDDESYS_ITEM_FORMATS	A list of DDE format numbers that the server can render.
SZDDESYS_ITEM_HELP	A text description of the server's DDE services.
SZDDESYS_ITEM_PROTOCOLS	A list of protocol names the server supports. A *protocol* is a set of DDE execute commands, each having a standard meaning.
SZDDESYS_ITEM_RESTART	A string that a client can pass to DosExecPgm to invoke a server that is not running.
SZDDESYS_ITEM_RTNMSG	Supporting detail for the most recently issued WM_DDE_ACK message. (This is useful when more than 8 bits of application-specific return code are required.)
SZDDESYS_ITEM_SECURITY	A security-sensitive server application. Any client can initiate a conversation with a security-sensitive server, but the server responds only to the *Security* topic. Typically, the server requires a password from the client before any further data exchange can take place.
SZDDESYS_ITEM_STATUS	An indication of the current status of the server.
SZDDESYS_ITEM_SYSITEMS	A list of the items supported under the system topic by this server.
SZDDESYS_ITEM_TOPICS	A list of the topics currently supported by the application. (This can vary from moment to moment).

Individual elements of lists should be delimited by tabs (the DDEFMT_TEXT format).

DDE Initiation

A client application initiates a DDE conversation by calling the WinDdeInitiate function, specifying the server application-name string and the topic-name string. WinDdeInitiate fills a DDEINIT structure with the specified strings, then sends a WM_DDE_INITIATE message to all frame windows that have HWND_DESKTOP as their parent. The message contains the handle of the client application and a pointer to the DDEINIT structure. The DDEINIT structure has the following form:

```
typedef struct _DDEINIT {
  ULONG   cb;
  PSZ     pszAppName;
  PSZ     pszTopic;
  USHORT  usConvContext;
} DDEINIT;
```

Because the message is sent rather than posted, WinDdeInitiate requires all the recipients of the message to respond to the message before returning control to the client application. Figure 32-2 illustrates the initial flow of a DDE conversation.

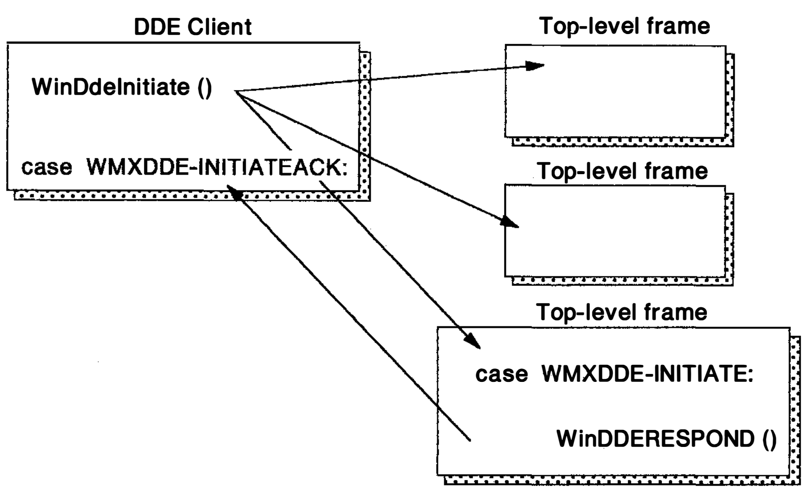


Figure 32-2. Initiating a DDE Conversation

Any potential server must contain a server window. A *server window* is a top-level frame window that has been subclassed to retrieve and process WM_DDE_INITIATE messages. For detailed information about window classes, see Chapter 3, "Window Classes" on page 3-1. When a server window retrieves WM_DDE_INITIATE, it examines the application-name and topic-name strings in the DDEINIT structure. If the application-name string matches, and the server supports the requested topic, the server acknowledges the client's request.

Either the application-name or topic-name string can be null. If the application-name string is null, all servers check the topic-name string. Each server that supports the topic sends a separate acknowledgment to the client. If the topic-name string is null, the server sends an acknowledgment for each supported topic. Using null strings, a client can obtain the names of all the active servers in the system or the names of all the topics a server supports.

The following code fragment shows how servers respond to WM_DDE_INITIATE messages:

```
If (specific app requested and server is instance of app) or
    (specific app not requested) {

     If (specific topic requested)
          If (server can support topic)

               acknowledge the requested topic

     else
          acknowledge each supported topic
}
```

A server acknowledges its support of a specific topic by calling the WinDdeRespond function, specifying the handle of its server window, its application name, and the name of the supported topic. WinDdeRespond fills a DDEINIT structure with the specified strings, then sends a WM_DDE_INITIATEACK message to the client. The message contains the handle of the server window and a pointer to the DDEINIT structure. The client examines the topic-name string in the DDEINIT structure and decides whether to begin a transaction on the topic.

It is legitimate for two applications that agree on some unspecified protocol and that exchange window handles by some means, to use DDE messages on those window handles without going through an initiate sequence.

An application typically does not need to fill in a DDEINIT structure; the WinDdeInitiate and WinDdeRespond functions automatically fill the structure. However, applications must extract the application name and topic name from the DDEINIT structure when receiving a WM_DDE_INITIATE or WM_DDE_INITIATEACK message.

Shared-Memory Object

After the client initiates a conversation, the client interacts with the server by issuing transactions. A *transaction* is a client's request that the server perform a particular action.

To issue a transaction, the client allocates a shared-memory object, writes data about its request to the object, then calls the WinDdePostMsg function to post a transaction message to the server. The transaction message contains the client-window handle and a pointer to the shared-memory object. When the server receives the message, it uses the pointer to access the shared-memory object.

The server responds by allocating a shared-memory object, writing its response to the object, then calling WinDdePostMsg to post a response message to the client. The response message contains the server-window handle and a pointer to the shared-memory object.

A DDESTRUCT structure occupies the first part of the memory object. Next comes the item-name string, followed by the actual data being exchanged. The offset fields of the DDESTRUCT structure must be set to point to the name string and the beginning of the data. The **cbData** field also must be set to indicate the number of bytes of data.

The sender of a DDE transaction message must allocate a shared-memory object using the DosAllocSharedMem function, then call the DosGiveSharedMem function to share the object with the receiving application. To share an object, the sender must know the process identifier of the recipient. The process identifier can be obtained by calling the WinQueryWindowProcess function for the recipient's window handle.

The sender should not try to access the object after sending it to the recipient in a DDE message. After posting a transaction message, the WinDdePostMsg function automatically frees the shared-memory object from the sender' virtual address space. An application need not call DosFreeMem for this purpose. However, the recipient should call DosFreeMem when it is finished using the object.

Transaction Status Flags

DDE client and server applications can specify status flags in the DDESTRUCT structure. These flags are constant values that applications use to control various aspects of a DDE transaction. They can be combined in the *fsStatus* word of the DDESTRUCT structure by using the OR operator. Following is a list of the DDE status flags:

Table 32-2. DDE Status Flags

Flag	Description
DDE_FACK	Indicates a positive acknowledgment.
DDE_FACKREQ	Requests an acknowledgment from the receiving application.
DDE_FAPPSTATUS	Indicates that the upper 8 bits of the status word are used for application-specific data.
DDE_FBUSY	Indicates that the application received a request but cannot respond because it is busy filling an earlier request.
DDE_FNODATA	Indicates that no data is to be transferred in response to the WM_DDE_ADVISE message.
DDE_FRESERVED	Reserved; must be 0.
DDE_FRESPONSE	Indicates a response to a WM_DDE_REQUEST message.
DDE_NOTPROCESSED	Indicates that the message received is not supported.

Transaction and Response Messages

DDE applications use the WinDdePostMsg function to communicate during data-exchange transactions. A client application posts transaction messages to a server. The server application responds by posting acknowledgment messages to the client.

Transaction and acknowledgment messages have the same structure. The first message parameter contains the handle of the sending window; the second contains a pointer to the shared-memory object that contains message information.

The DDE protocol defines five transaction types: advise, unadvise, request, poke, and execute. These transactions are permitted only within an exchange begun by using the WM_DDE_INITIATE message. Each transaction type has a corresponding message that a client uses to initiate the transaction with a server. These messages include WM_DDE_ADVISE, WM_DDE_UNADVISE, WM_DDE_REQUEST, WM_DDE_POKE, and WM_DDE_EXECUTE.

A server acknowledges a transaction message by posting a WM_DDE_ACK message to the client. The client must examine the status field of the DDESTRUCT structure to determine whether the response is positive or negative.

A server application posts a WM_DDE_DATA message to the client to indicate that requested data is available. If the status bit of the DDESTRUCT structure has the DDE_FACKREQ flag set, the client must acknowledge receipt of the data by sending a WM_DDE_ACK message to the server.

The fifth parameter of the WinDdePostMsg function is a flag used to specify whether to try to post a message again if the first attempt failed because the destination queue was full. If the retry flag is set, WinDdePostMsg posts the message at 1-second intervals until the message is posted successfully.

The following sections explain the five basic types of DDE transactions and the messages involved with each.

Request and Poke Transactions

A client application can use the DDE protocol to obtain a data item from a server (WM_DDE_REQUEST) or to submit a data item to a server (WM_DDE_POKE).

The client posts a WM_DDE_REQUEST message to the server, specifying an item and format by allocating a shared-memory object, filling in a DDESTRUCT structure, and passing the structure to the WinDdePostMsg function.

If the server is unable to satisfy the request, it sends the client a negative WM_DDE_ACK message. If the server can satisfy the request, it renders the item in the requested format, includes it with a DDESTRUCT structure in a shared-memory object, and posts a WM_DDE_DATA message to the client.

Upon receiving a WM_DDE_DATA message, the client processes the data item. At the beginning of the shared-memory object, the DDESTRUCT structure contains a status word indicating whether the sender requested an acknowledgment message. If the DDE_FACKREQ bit of the status word is set, the client must send the server a positive WM_DDE_ACK message.

Upon receiving a negative WM_DDE_ACK message, the client can ask for the same item again, specifying a different DDE format. Typically, a client first asks for the most complex format it can support, then steps down, if necessary, through progressively simpler formats, until it finds one the server can provide.

Advise and Unadvise Transactions

A client application can use DDE to establish a link to an item in a server application. When such a link is established, the server sends periodic updates about the linked item to the client (typically, whenever the data associated with the item in the server application has changed). A permanent *data stream* is established between the two applications and remains in place until it is explicitly disconnected.

The client sends the server a WM_DDE_ADVISE message to set up the data link. The advise message contains a shared-memory pointer containing a DDESTRUCT structure with the item name, format information, and status information.

If the server has access to the requested item and can render it in the desired format, the server records the new link, then sends the client a positive WM_DDE_ACK message. Until the client issues a WM_DDE_UNADVISE message,

the server sends data messages to the client every time a change occurs in the source data associated with the item in the server application.

If the server is unable to satisfy the request, it sends the client a negative WM_DDE_ACK message.

When a link is established with the DDE_FNODATA status bit cleared, the client is sent the data each time the data changes. In such cases, the server renders the new version of the item in the previously specified format and posts a WM_DDE_DATA message to the client.

When the client receives a WM_DDE_DATA message, it extracts data from the shared-memory object by using the DDESTRUCT structure at the beginning of the object. If the DDE_FACKREQ status bit in the status word of the DDESTRUCT structure is set, the client must post a positive WM_DDE_ACK message to the server.

When a link is established with the DDE_FNODATA status flag set, a notification, not the data itself, is posted to the client each time the data changes. In this case, the server does not render the new version of the item when the source data changes, but simply posts a WM_DDE_DATA message with 0 bytes of data and the DDE_FNODATA status flag set.

The client can request the latest version of the data by performing a regular one-time WM_DDE_REQUEST transaction, or it can simply ignore the data-change notice from the server. In either case, if the DDE_FACKREQ status bit is set, the client should send a positive WM_DDE_ACK message to the server.

When a client sends a WM_DDE_ADVISE message on a topic/item pair that is already engaged in an advise loop but has a different format specified, the server interprets this as a request to add an advise loop with the given format requested. Therefore, several advise loops can exist for a given topic/item pair. If a server does not support this extent of advise loops, it rejects the advise request.

Correspondingly, when a server receives a WM_DDE_UNADVISE message, the server must compare the format field with the current format of the advise loop. Only if the specified format is 0 (*wildcard*) or matches an active advise loop does the server stop the advise loop and return a positive acknowledgment.

To terminate a specific item link, the client posts a WM_DDE_UNADVISE message to the server. The server ensures that the client currently has a link to the specified item in this exchange. If the link exists, the server sends a positive WM_DDE_ACK message to the client and no longer sends updates on the item in this exchange. If the server has no such link, it sends a negative WM_DDE_ACK message.

To terminate all links for a particular exchange, the client application posts a WM_DDE_UNADVISE message with a null item name to the server. The server ensures that the exchange has at least one link currently established. If so, the server posts a positive WM_DDE_ACK message to the client, and no longer sends any updates in the exchange. If the server has no links in the exchange, it posts a negative WM_DDE_ACK message.

Execute Transaction

A PM application can use the DDE protocol to cause a command or series of commands to be executed in another application. Such remote executions are performed by the WM_DDE_EXECUTE transaction.

To execute a remote command, the client application posts to the server a WM_DDE_EXECUTE message containing a pointer to a shared-memory object that contains a DDESTRUCT structure and a command string.

The server attempts to execute the specified string according to some agreed-upon protocol. If successful, the server posts a positive WM_DDE_ACK message to the client; if unsuccessful, a negative WM_DDE_ACK message is posted.

DDE Termination

At any time, either the client or the server may terminate an exchange by issuing a WM_DDE_TERMINATE message. Similarly, both the client application and server application must be able to receive a WM_DDE_TERMINATE message at any time.

An application must end its exchanges before terminating. The application posts a WM_DDE_TERMINATE message with a NULL shared-memory pointer. A WM_DDE_TERMINATE message stops all transactions for a given exchange.

The WM_DDE_TERMINATE message means that the sender will send no further messages in that exchange and that the recipient can destroy its DDE window. The recipient always must send a WM_DDE_TERMINATE message promptly in response; it is not permissible to send a negative, busy, or positive WM_DDE_ACK message instead.

If the sender of the original termination request receives any other message before the WM_DDE_TERMINATE message arrives from the recipient of the request, no response should be sent to this other message. The sender of the other message might have destroyed the window to which the response would be sent.

Unique Data Formats

Whenever an application exchanges data by using the DDE protocol, it must specify the format of the data in the **usFormat** field of the DDESTRUCT structure. The system-defined standard format for exchanging text data is DDEFMT_TEXT. Applications also can use the following constant names to specify the format of data to be exchanged.

Table 32-3 (Page 1 of 2). DDE Data Formats

Format	Description
SZFMT_BITMAP	Specifies that the data is a bit map.
SZFMT_CPTEXT	Specifies text whose format is defined by a CPTEXT structure. Applications can use this format to pass multiple-language strings without changing the conversation context.
SZFMT_DIF	Specifies that the data is in Data Image Format (DIF).
SZFMT_DSPBITMAP	Specifies that the data is a bit-map representation of a private data format.
SZFMT_DSPMETAFILE	Specifies that the data is a metafile representation of a private data format.

Table 32-3 (Page 2 of 2). DDE Data Formats	
Format	**Description**
SZFMT_DSPMETAFILEPICT	Specifies that the data is a metafile picture representation of a private data format.
SZFMT_DSPTEXT	Specifies that the data is a text representation of a private data format.
SZFMT_LINK	Specifies that the data is in link-file format.
SZFMT_METAFILE	Specifies that the data is a metafile.
SZFMT_METAFILEPICT	Specifies that the data is a metafile picture defined by an MFP structure.
SZFMT_OEMTEXT	Specifies that the data is in OEM Text format.
SZFMT_PALETTE	Specifies that the data is in palette format.
SZFMT_SYLK	Specifies that the data is in Synchronous Link format.
SZFMT_TEXT	Specifies that the data is an array of text characters. These characters can include newline characters to indicate linebreaks. The NULL character indicates the end of the text data.
SZFMT_TIFF	Specifies that the data is in Tag Image File Format (TIFF).

Applications can define their own data formats. Each nonstandard DDE format must have a unique identification number. The application must register the name of the format in the system atom table, receiving an identification number for that format name. Other applications that have the name of the format also can query the system atom table for the format's identification number. This method ensures that all applications use the same atom to identify a format. For information on how to register a nonstandard DDE format with the system atom table, see Chapter 35, "Atom Tables" on page 35-1.

Synchronization Rules

A window processing DDE requests from another window must process them strictly in the order in which the requests were received.

A window does not need to apply this first-in first-out (FIFO) rule between requests from different windows—that is, it may provide asynchronous support for multiple processes. For example, a window might have the following requests in its queue:

1. Request message from window x
2. Request message from window y
3. Request message from window x.

The window must process request message 1 before request message 3, but it does not have to process request message 2 before request message 3. If y has a lower priority than x, the window follows the order 1, 3, 2.

If a server is unable to process an incoming request because it is waiting for an external process, it must post a busy WM_DDE_ACK message to the client, to prevent deadlock. A busy WM_DDE_ACK message also can be sent if the server is unable to process an incoming request quickly.

Language-Sensitive DDE Applications

DDE applications written for the international market must be able to exchange data in several different languages. The CONVCONTEXT structure, along with the WinDdeInitiate2 and WinDdeRespond2 functions, provide this support.

A language-sensitive DDE application defines the context of a conversation by filling a CONVCONTEXT structure with the appropriate country code and codepage identifiers. The CONVCONTEXT structure also contains a context flag. If this flag is set to DDECTXT_CASESENSITIVE, applications must compare strings in a case-sensitive manner.

Language-sensitive DDE applications use the WinDdeInitiate2 and WinDdeRespond2 functions to establish a DDE conversation. These functions pass the same arguments as their counterparts, WinDdeInitiate and WinDdeRespond. The difference is that WinDdeInitiate2 and WinDdeRespond2 also pass a pointer to a CONVCONTEXT structure.

Using Dynamic Data Exchange

This section explains how to perform the following tasks:

- Create a shared-memory object for DDE.
- Send positive acknowledgment messages.
- Send negative acknowledgment messages.
- Perform a one-time data transfer.
- Establish a permanent data link.
- Execute commands in a remote application.
- Terminate a DDE conversation.

Creating a Shared-Memory Object for DDE

The following code fragment shows a function that creates a shared-memory object for a DDE transaction. The function parameters include the destination window for the DDE message, item name for the transaction, status word, format of the data, actual data to be transferred (if any), and the length of the data. The object allocated by this function must be big enough to hold the DDESTRUCT structure, item name, and the actual data to be transferred. The function returns a pointer (PDDESTRUCT) to a shared-memory object that is ready to post as part of a DDE message.

```
PDDESTRUCT MakeDDEObject(HWND hwnd, PSZ pszItemName,
                         USHORT  fsStatus,USHORT usFormat,
                         PVOID pabData,USHORT usDataLen)
{
    PDDESTRUCT pddes;          /* Pointer to DDESTRUCT     */
    ULONG      usItemLen;      /* Length of item name      */
    PULONG     pulSharedObj;   /* Pointer to shared object */
    PID        pid;            /* Process ID               */
    TID        tid;            /* Thread ID                */

    if (pszItemName != NULL)
        usItemLen = strlen(pszItemName) + 1;
    else
        usItemLen = 1;

    if (!(DosAllocSharedMem((PVOID)&pulSharedObj, NULL,
            sizeof(DDESTRUCT) + usItemLen + usDataLen,
            PAG_COMMIT | PAG_READ | PAG_WRITE | OBJ_GIVEABLE))) {


        WinQueryWindowProcess(hwnd,&pid,&tid);

        DosGiveSharedMem(&pulSharedObj,pid,PAG_READ|PAG_WRITE);


        /* Initialize DDESTRUCT. */
        pddes = (PDDESTRUCT) pulSharedObj;
        pddes->cbData = (LONG) usDataLen;
        pddes->fsStatus = fsStatus;
        pddes->usFormat = usFormat;
        pddes->offszItemName = sizeof(DDESTRUCT);
        if ((usDataLen) && (pabData != NULL))
            pddes->offabData = sizeof(DDESTRUCT) + usItemLen;
        else
            pddes->offabData = 0;

        /* Copy item name immediately following DDESTRUCT. */
        if (pszItemName != NULL)
            StringCopy(DDES_PSZITEMNAME(pddes), pszItemName);
        else
            StringCopy(DDES_PSZITEMNAME(pddes), "");

        /* Copy data immediately following item name. */
        if (pabData != NULL)
            DataCopy(DDES_PABDATA(pddes), pabData, usDataLen);

        return (pddes);
    }
    return ((PDDESTRUCT) NULL);
}
```

This function is used in many examples in the following sections to demonstrate the creation of DDE shared-memory objects. You might want to define a similar function in your own programs as well.

Sending a Positive Acknowledgment

You can send a positive acknowledgment by posting a WM_DDE_ACK message with the DDE_FACK and DDE_FRESPONSE flags set in the status word of the DDEINIT structure. The following code fragment is an example of a positive acknowledgment message:

```
HWND hwndDest,hwndSource;
PDDESTRUCT pddeStruct;

pddeStruct = MakeDDEObject(hwndDest,     /* Handle of destination  */
    "BTRX",                              /* Item name              */
    DDE_FACK | DDE_FRESPONSE,            /* Status flags           */
    DDEFMT_TEXT,                         /* Data format            */
    NULL,                                /* No data for request    */
    0);                                  /* Data length            */

WinDdePostMsg(hwndDest,                  /* Handle of destination  */
    hwndSource,                          /* Handle of source       */
    WM_DDE_ACK,                          /* Message                */
    pddeStruct,                          /* Shared-memory  pointer */
    1);                                  /* Retry                  */
```

Sending a Negative Acknowledgment

You can send a negative acknowledgment by posting a WM_DDE_ACK message with the DDE_NOTPROCESSED flag set in the status word of the DDEINIT structure. The following code fragment is an example of a negative acknowledgment message:

```
HWND hwndDest,hwndSource;
PDDESTRUCT pddeStruct;

pddeStruct = MakeDDEObject(hwndDest,     /* Handle of destination  */
    "BTRX",                              /* Item name              */
    DDE_NOTPROCESSED,                    /* Status flags           */
    DDEFMT_TEXT,                         /* Data format            */
    NULL,                                /* No data for request    */
    0);                                  /* Data length            */

WinDdePostMsg(hwndDest,                  /* Handle of destination  */
    hwndSource,                          /* Handle of source       */
    WM_DDE_ACK,                          /* Message                */
    pddeStruct,                          /* Shared-memory pointer  */
    1);                                  /* Retry                  */
```

If an application is busy when it receives a DDE message, it can post a WM_DDE_ACK message with the DDE_FBUSY flag set.

Performing a One-Time Data Transfer

A client application posts a WM_DDE_REQUEST or WM_DDE_POKE message to perform a one-time data transfer with a server application. The item-name portion of the shared-memory object passed with the message contains the name of the desired item. When the client posts a WM_DDE_POKE message, the data portion of the object contains the data being sent to the server.

The following code fragment is an example of a request transaction:

```
HWND hwndServer,hwndClient;
PDDESTRUCT pddeStruct;

pddeStruct = MakeDDEObject(hwndServer,  /* Handle of server      */
    "BTRX",                             /* Item name             */
    0,                                  /* Status flags          */
    DDEFMT_TEXT,                        /* Data format           */
    NULL,                               /* No data for request   */
    0);                                 /* Data length           */

WinDdePostMsg(hwndServer,               /* Handle of server      */
    hwndClient,                         /* Handle of client      */
    WM_DDE_REQUEST,                     /* Message               */
    pddeStruct,                         /* Shared-memory pointer */
    1);                                 /* Retry                 */
```

If the server can satisfy the request, it renders the item in the requested format and includes it, with a DDESTRUCT structure, in a shared-memory object and posts a WM_DDE_DATA message to the client, as shown in the following code fragment:

```
HWND hwndClient,hwndServer;
PDDESTRUCT pddeStruct;
USHORT usDataLen;
PVOID pabData;

pddeStruct = MakeDDEObject(hwndClient,  /* Handle of client      */
    "BTRX",                             /* Item name             */
    0,                                  /* Status flags          */
    DDEFMT_TEXT,                        /* Data format          */
    pabData,                            /* Pointer to data       */
    usDataLen);                         /* Data length           */

WinDdePostMsg(hwndClient,               /* Handle of client      */
    hwndServer,                         /* Handle of server      */
    WM_DDE_DATA,                        /* Message               */
    pddeStruct,                         /* Shared-memory pointer */
    1);                                 /* Retry                 */
```

Establishing a Permanent Data Link

The client posts a WM_DDE_ADVISE message to the server to set up a permanent data link. The advise message contains a shared-memory pointer containing a DDESTRUCT structure with the item name, format information, and status information, as shown in the following code fragment:

```
HWND hwndServer,hwndClient;
PDDESTRUCT pddeStruct;

pddeStruct = MakeDDEObject(hwndServer,   /* Handle of server      */
    "BTRX",                              /* Item name             */
    DDE_FACKREQ,                         /* Status flags          */
    DDEFMT_TEXT,                         /* Data format           */
    NULL,                                /* No data for advise    */
    0);                                  /* Data length           */

WinDdePostMsg(hwndServer,                /* Handle of server      */
    hwndClient,                          /* Handle of client      */
    WM_DDE_ADVISE,                       /* Message               */
    pddeStruct,                          /* Shared-memory pointer */
    1);                                  /* Retry                 */
```

When a link is established with the DDE_FNODATA status flag set, a notification, not the data itself, is posted to the client each time the data changes. In this case, the server does not render the new version of the item when the source data changes, but simply posts a WM_DDE_DATA message with 0 bytes of data and the DDE_FNODATA status flag set, as shown in the following code fragment:

```
HWND hwndServer,hwndClient;
PDDESTRUCT pddeStruct;

pddeStruct = MakeDDEObject(hwndClient,   /* Handle of client      */
    "BTRX",                              /* Item name             */
    DDE_FNODATA,                         /* Status flags          */
    DDEFMT_TEXT,                         /* Data format           */
    NULL,                                /* No data               */
    0);                                  /* Data length           */

WinDdePostMsg(hwndClient,                /* Handle of client      */
    hwndServer,                          /* Handle of server      */
    WM_DDE_DATA,                         /* Message               */
    pddeStruct,                          /* Shared-memory pointer */
    1);                                  /* Retry                 */
```

The client terminates a data link by posting a WM_DDE_UNADVISE message to the server, as shown in the following code fragment:

```
HWND hwndServer,hwndClient;
PDDESTRUCT pddeStruct;

pddeStruct = MakeDDEObject(hwndServer,  /* Handle of server      */
    "BTRX",                             /* Item name             */
    DDE_FACKREQ,                        /* Status flags          */
    DDEFMT_TEXT,                        /* Data format           */
    NULL,                               /* No data for unadvise  */
    0);                                 /* Data length           */

WinDdePostMsg(hwndServer,               /* Handle of server      */
    hwndClient,                         /* Handle of client      */
    WM_DDE_UNADVISE,                    /* Message               */
    pddeStruct,                         /* Shared-memory pointer */
    1);                                 /* Retry                 */
```

Executing Commands in a Remote Application

To execute a remote command, the client application posts to the server a WM_DDE_EXECUTE message containing a pointer to a shared-memory object that contains a DDESTRUCT structure and a command string, as shown in the following code fragment:

```
HWND hwndServer,hwndClient;
PDDESTRUCT pddeStruct;
PVOID pabData;
USHORT usDataLen;

pddeStruct = MakeDDEObject(hwndServer,  /* Handle of server         */
    "BTRX",                          /* Item name                */
    DDE_FACKREQ,                     /* Status flags             */
    DDEFMT_TEXT,                     /* Data format              */
    pabData,                         /* Pointer to command string */
    usDataLen);                      /* Data length              */

WinDdePostMsg(hwndServer,            /* Handle of server         */
    hwndClient,                      /* Handle of client         */
    WM_DDE_EXECUTE,                  /* Message                  */
    pddeStruct,                      /* Shared-memory pointer    */
    1);                              /* Retry                    */
```

Terminating a DDE Conversation

At any time, either the client or the server may terminate a DDE conversation by posting a WM_DDE_TERMINATE message, as shown in the following code fragment.

```
HWND hwndDest,hwndSource;

WinDdePostMsg(hwndDest,      /* Handle of destination    */
    hwndSource,              /* Handle of source         */
    WM_DDE_TERMINATE,        /* Message                  */
    NULL,                    /* No shared-memory pointer */
    1);                      /* Retry                    */
```

Summary

The following tables describe the functions, structures and messages associated with the DDE protocol.

Table 32-4. Window Procedure Syntax

Function Name	Description
WinDdeInitiate	Issued by a client application to one or more other applications, to request initiation of a dynamic data exchange conversation with a national language conversation context.
WinDdeInitiate2	Passes the same arguments as WinDdeInitiate, but also passes a pointer to a CONVCONVERT structure.
WinDdePostMsg	Issued by an application to post a message to another application with which it is carrying out a dynamic data exchange conversation with a national language conversation.
WinDdeRespond	Issued by a server application to indicate that it can support a dynamic data exchange conversation on a particular topic with a national language conversation context.
WinDdeRespond2	Passes the same arguments as WinDdeRespond, but also passes a pointer to a CONVCONVERT structure.

Table 32-5. DDE Structures

Structure Name	Description
CONVCONTEXT	Dynamic data exchange conversation context structure.
DDEINIT	Dynamic data exchange initiation structure.
DDESTRUCT	Dynamic data exchange control structure.

Table 32-6. DDE Messages

Message	Description
WM_DDE_ACK	Notifies an application of the receipt and processing of a WM_DDE_EXECUTE, WM_DDE_DATA, WM_DDE_UNADVISE, or WM_DDE_POKE message, and in some cases, a WM_DDE_REQUEST message.
WM_DDE_ADVISE	Requests the receiving application to supply an update for a data item whenever it changes.
WM_DDE_DATA	Notifies a client application of the availability of data.
WM_DDE_EXECUTE	Posts a string to a server application to be processed as a series of commands.
WM_DDE_INITIATE	Sent by an application to one or more other applications to request initiation of a conversation.
WM_DDE_INITIATEACK	Sent by a server application in response to a WM_DDE_INITIATE message, for each topic that the server application wishes to support.
WM_DDE_POKE	Requests an application to accept an unsolicited data item.
WM_DDE_REQUEST	Posted from client to server, to request that the server provide a data item to the client.
WM_DDE_TERMINATE	Posted by either application participating in a DDE conversation to terminate that conversation.
WM_DDE_UNADVISE	Posted by a client application to a server application to indicate that the specified item should be updated no longer.

Chapter 33. Direct Manipulation

Direct manipulation is the act of moving graphical representations (OS/2 icons, for example) around the screen using a pointing device, such as a mouse. This chapter explains how to use direct manipulation in PM applications.

About Direct Manipulation

The direct manipulation protocol enables the user to visually drag an object in a window and drop it on another object in a window. *Dragging* is moving an object as though it were attached to the pointer; it is performed by holding down the select button and moving the pointer. *Dropping* is fixing the position of the dragged object by releasing the select button on the pointer. This causes interaction (data exchange) between the window from which the object was dragged and the window containing the object being dropped on.

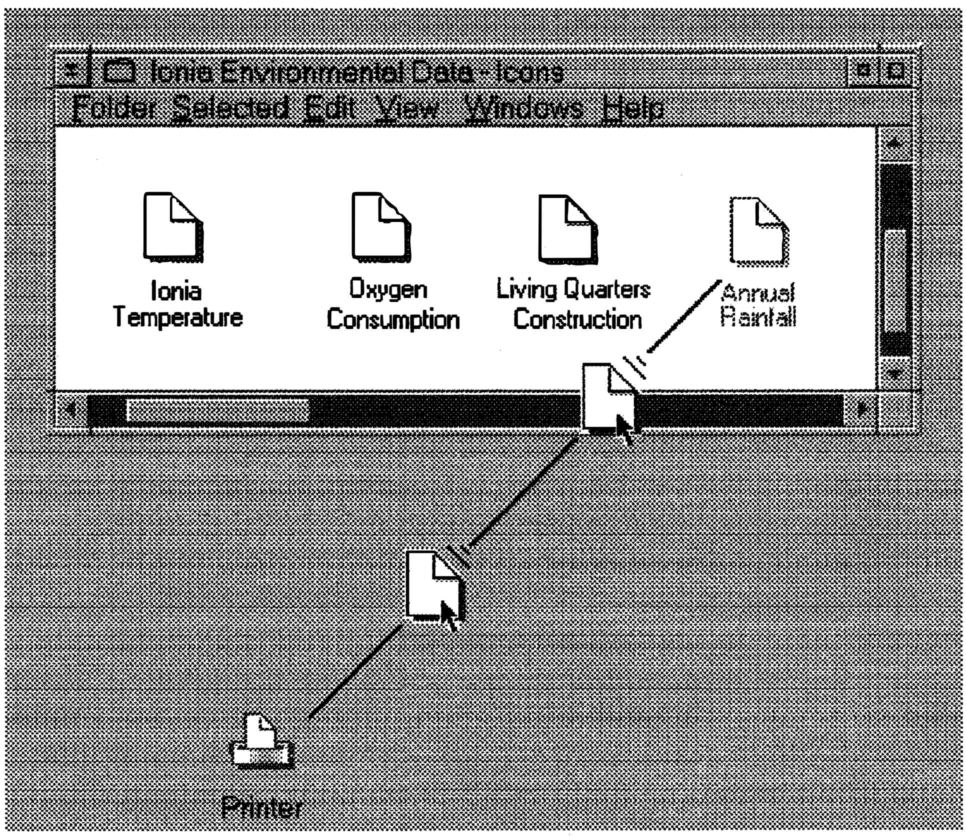


Figure 33-1. Dragging Data to a Printer

The window containing the dragged object is referred to as the *source*. The window containing the object that was dropped on is referred to as the *target*. The source and target can be the same window, different windows within the same application, or windows belonging to different applications. The dragged object can be the only visible object in the source window, or it can be one of many objects. The target object can be the only visible object in the target window, or it can be one of many objects. A source or target window that contains multiple objects is referred to as a *container window*.

The data exchange that occurs between the source and target after a direct manipulation operation enables applications in the system that support the protocol to integrate easily, while providing a simple user interface.

Using Direct Manipulation in an Application

The application's responsibilities during a direct manipulation operation vary, depending on whether the application's window procedure is acting as the source or the target of the operation.

Writing a Source Application

The source is responsible for starting a direct manipulation operation. Startup can be accomplished only with a pointing device, such as a mouse. The operation starts when the application detects that a select button has been pressed and the pointing device has moved. Dragging continues until terminated, usually when the button is released.

Although the direct manipulation protocol lets the application use any button for dragging, it is recommended that the system-defined direct manipulation button (*drag button*) be used for direct manipulation operations.

"Two-Object Drag" on page 33-12 shows the sequence of API functions and the message flow for a typical direct manipulation operation. The flow illustrates a two-object drag—from Application 1 to Application 3—dragging over Application 2.

The direct manipulation operation was started by the source window procedure after the user selected the object (or objects) to be manipulated and the source received a WM_BEGINDRAG message.

The source has the following responsibilities in preparing for the actual drag of the object or objects across the screen:

- Allocate and initialize the DRAGINFO structure that will convey the necessary information about each object to the target.
- Initialize a set of DRAGIMAGE structures that describe the image to be displayed during the drag operation.
- Make the type of each object being directly manipulated known to the system.
- Make the rendering mechanism and format for each object known to the system. For detailed information, see "Standard Rendering Mechanisms" on page 33-18.
- Make the suggested name of the object at the target known to the system.
- Make the name of the container or folder containing the source object known to the system.
- Make the name of the object at the source known to the system.
- Make the *true* type of each object being directly manipulated known to the system.
- Make the native rendering mechanism and format for each object known to the system.

To prepare for the drag operation, the source must invoke DrgAllocDraginfo to allocate memory for the DRAGINFO structure. DrgAllocDraginfo initializes the DRAGINFO structure as follows:

cbDraginfo	The size, in bytes, of the entire DRAGINFO structure, including the DRAGITEM array.
cbDragitem	The size, in bytes, of each DRAGITEM structure.
usOperation	Initialized to DO_DEFAULT.
xDrop and **yDrop**	Initialized to the current mouse-pointer location, in desktop coordinates.
cditem	Initialized to the count of objects being dragged, as specified in DrgAllocDraginfo.

The source then completes the initialization of each DRAGITEM structure, as appropriate, for each of the objects to be dragged. This is accomplished by using the DrgSetDragitem function, or by obtaining a pointer to each DRAGITEM structure with DrgQueryDragitemPtr, and initializing it directly.

The first step the source takes to initialize the DRAGITEM structure is to create the appropriate drag string handles. String handles must be created for:

- Object type or types
- Supported rendering mechanisms and formats for the object
- Suggested name of the object at the target
- Name of the container holding the object (whether a container or folder)
- Name of the object at the source when the source allows the target to carry out the operation for the object.

Then the balance of the DRAGINFO structure for that object can be initialized as appropriate.

Type: To directly manipulate an object, both the source and the target must know the object type and understand that type. The **hstrType** field in the DRAGITEM structure conveys this information for each object being dragged. The type is represented by a string handle. The target should check to see if it understands the type prior to allowing the user to drop the object.

Several DTYP_* constants are defined as *notational conveniences* for common types of data. An application can extend these types by defining its own character strings, then creating string handles for them using the DrgAddStrHandle function.

True Type: The *true type* of an object is the type that most accurately describes the object. For example, the input to a C compiler could have the type *Plain Text* (DRT_TEXT), but would be more accurately described as *C Code* (DRT_C). *C Code* would be the true type of this object.

Multiple types can be conveyed by using a comma to separate strings, as follows:

```
type,type
```

The true type should appear first in the list of types, so the type string for the example object would be "C Code, Plain Text."

Rendering Mechanism and Format: The rendering mechanism and format is a string handle. The string takes the form:

```
"elem {,elem,elem...}"
```

where *elem* is an ordered pair in the form:

```
"<mechanism,format>"
```

or a cross product in the form:

```
"(mechanism{,mechanism...}) X (format{,format...})"
```

Multiple cross products are permitted in a single rendering mechanism and format string handle, as are combinations of ordered pairs and cross products. When cross-product notation is used, the rendering mechanism is the left operand. When ordered-pair notation is used, the rendering mechanism is the left element in the ordered pair.

The rendering mechanism represents the way in which you want to exchange the data, for example *dynamic data exchange (DDE)*. The rendering format identifies the actual format of the data, for example, *text*. To exchange data, both the source and target must know how to communicate with each other through the rendering mechanism and understand the particular format of the data. The target should verify that it understands the rendering mechanism and format before allowing the user to drop the object or objects. The rendering mechanism and format are passed as a string handle in the DRAGITEM structure. The string handle must be created using the DrgAddStrHandle function.

Several constants are defined for common rendering mechanisms and formats. An application can extend these by defining its own "<*mechanism,format*>" strings and creating string handles for these using the DrgAddStrHandle function.

For example, if an application understands and can generate an LU 6.2 data stream, it can define its own rendering format, "DRF_LU62," and use it in direct manipulation operations. If an application wishes to use its own rendering mechanisms or formats to communicate with other applications, it should publish the protocol for the mechanisms, the format of the data streams, or both.

Native Rendering Mechanism and Format: The native rendering mechanism and format of the object is the mechanism that most naturally conveys the data and its current format. For example, the native rendering mechanism and format for:

- A C source file might be <DRM_OS2FILE,CF_OEMTEXT>
- A spreadsheet file might be <DRM_OS2FILE,CF_SYLK>

In some direct-manipulation operations, it might be possible for the target to carry out the necessary action on the source object without the source's participation. However, this would be possible only when the target understands both the true type and the native rendering mechanism and format of the object. Even when the target is not performing the necessary action on the source object, it is still important to know the native rendering mechanism and format. In determining the rendering mechanism and format to be used in the data exchange after the drop, the target might select the native format, since, generally, performance is better when the native rendering mechanism and format is used.

The native rendering mechanism and format is conveyed to the target by making it the first ordered pair, or the first ordered pair to result from a cross product, in the list of rendering mechanisms and formats passed in the DRAGINFO structure.

Suggested Name at Target: When dragging an object, for example a file, from one container to another, it is important to know the name the object should have at the target. This may or may not be the same name it had at the source. This name enables the target to check if another object with the same name already exists at the target and to take the appropriate action. For example, a target container might not allow the user to drop the object, or objects, if an object by that same name exists at the target.

Container Name: Sometimes it is necessary for a target container to be aware of the name of the source container. This name could carry some location information. For example, the default operation when dragging objects between containers is a *Move*. However, in the case of file folders on different drives, this default would be changed to a *Copy* operation. Thus, a file folder would fill this field with the drive and path information for the file. For example, (`A:\SUBDIR1\SUBDIR2\`). A database container, on the other hand, might fill this field with the fully qualified OS/2 file name of the database.

Source Name: In some direct-manipulation operations, it is possible for the target to perform the necessary action on the source object without the source's participation. If the source allows this, the target name should be filled in with the name of the source object. For example, a file folder would put the name of the source file into this field, such as (autoexec.bat). A database manager, on the other hand, might fill this field with some location information so the target could find a particular record or field within the database.

Dragging the Objects

Once initialization is complete, the source invokes the DrgDrag function to accomplish the direct manipulation operation. As the pointer moves around the screen, the system sends a DM_DRAGOVER message. The target window receiving the DM_DRAGOVER message responds with DOR_DROP if it understands the type and rendering formats of the objects being dragged, as well as the operation being performed. When a potential target cannot allow the objects to be dropped at this location in its window, it should respond with DOR_NODROP or DOR_NODROPOP. When a potential target cannot allow the objects to be dropped anywhere in its window, it should respond with DOR_NEVERDROP. This last case prevents multiple DM_DRAGOVER messages from being sent unnecessarily to a window when the pointer moves again or when the user presses another augmentation key.

To determine the proper reply to a DM_DRAGOVER message, the target gets information about the direct manipulation operation by using the DrgAccessDraginfo function. The DM_DRAGOVER message contains a pointer to the DRAGINFO structure. The target can access this structure with the DrgAccessDraginfo call, thus making all information about the direct manipulation operation available to the target window.

If the target responds to the DM_DRAGOVER message with DOR_NODROP, the system changes the image displayed to indicate that a drop is not allowed. When the user moves the pointer, or presses or releases an augmentation key, the system sends another DM_DRAGOVER message.

If a DM_DRAGOVER message receives a reply of DOR_NODROPOP, the system changes the displayed image to indicate that a drop is prohibited until the user moves the pointer outside the current target window or presses another augmentation key. When either of these events occurs, DrgDrag sends another DM_DRAGOVER message. If the user presses another augmentation key but has

not moved the pointer, a DM_DRAGOVER message is sent to the same window, giving it an opportunity to accept the drop for the new operation.

If DOR_NEVERDROP is returned from the DM_DRAGOVER message, further DM_DRAGOVER messages are not sent to the target until the pointer is moved outside of and back into the target window. A no-drop image will be displayed.

Application-Defined Drag Operations

This protocol defines a method for integrating two unrelated applications through direct manipulation. At times it may be useful for an application to define its own drag operation to facilitate functions between two windows in the same application, or between closely related applications. For example, an application implementing a keyboard remapping function may want to provide a method of redefining keys with direct manipulation. This application could define an operation whereby dragging one key to another exchanges the definitions of the two keys. The protocol provides the extendability to enable this kind of function.

Completing a Direct Manipulation Operation

The user can end a direct manipulation operation in one of three ways:

- Pressing the Esc key to cancel the operation
- Releasing the drag button when the pointer is over a target that cannot accept the drop.

  This action is equivalent to pressing the Esc key. When the pointer is over a target that can accept the drop, the target is informed of the drop, and the source is given the window handle of the target.
- Pressing the F1 key to request help.

  A DM_DROPHELP message is posted to the target. This enables the target to provide the user with assistance regarding:
  - What would happen if the user dropped the object or objects on that target
  - Why the target cannot accept a particular drop.

  The source sees this termination of the direct manipulation operation as a cancelation.

When the user drops the objects, a a DM_DROP message is sent to the target, providing it with the information necessary to process the objects that were dropped. The target application uses the information provided to exchange data with the source. The protocol to be used depends on the rendering mechanism specified for each object. It is the responsibility of the target to establish the appropriate conversation or conversations. It is the responsibility of the source to cooperate in the establishment of the necessary conversation or conversations to achieve the actual data exchange. After completing the direct manipulation operation, including the post-drop conversation with the source, the target uses DrgDeleteStrHandle or DrgDeleteDraginfoStrHandles to delete the string handles in the DRAGINFO structure, and DrgFreeDraginfo to release the storage.

DRAGDROP Sample Program

A sample program, DRAGDROP, is provided with the Toolkit to demonstrate the use of the direct manipulation protocol. DRAGDROP is a simple directory-navigation application. Two copies of this sample must be running to drag an object from one window to another.

When you start DRAGDROP, the contents of the root directory on drive C: are displayed in a list. Directories are displayed with a leading "/" character.

You can navigate downward in the directory tree by double clicking on a directory displayed in the list. You also can navigate downward by selecting **File** on the action bar, then **Open....** from the File pull-down menu. This dialog also can be used to navigate upward in the directory or to select another drive.

To move or copy files or directories from one directory window to another, use the select button (button 1 on your pointing device) to select one or more files or directories in the source application. Then press and hold the drag button (button 2), and drag the objects to the target application. Release the drag button when the pointer is over your intended target. The selected files will be moved or copied from the source directory to the target directory; the contents of the windows are updated automatically. The default operation is a Move, but you can change this to a Copy by pressing the Ctrl key along with the drag button.

Summary of Functions Used by the Source

The following table summarizes the functions a source would use in direct manipulation:

Table 33-1. Summary of Functions Used by the Source

Function Name	**Description**
DrgAddStrHandle	Creates a handle for an input string.
DrgAllocDraginfo	Allocates a DRAGINFO structure in shared memory.
DrgAllocDragTransfer	Allocates a specified number of DRAGTRANSFER structures from a single segment.
DrgDrag	Handles movement of the source-specified pointer around the screen. Provides visible feedback to the user.
DrgFreeDraginfo	Deallocates the memory associated with a DRAGINFO structure.
DrgSetDragitem	Initializes each object element in a DRAGINFO structure.

Writing a Target Application

The target in a direct manipulation operation is responsible for determining whether a particular set of objects can be dropped on it, and aids in providing the user with visible cues regarding the operation. A target is informed of the operation through messages sent to it as the pointer, provided by the source, is dragged across the screen.

When a set of objects is dropped on the target, the target is responsible for establishing the appropriate conversation or conversations with the source to accomplish the data transfer. The type of conversation for each object will be based on the rendering mechanism and format of the object being dropped.

Messages Sent to a Target Application

The following messages are sent to each window whose boundaries are crossed as the user drags the object or objects around the screen.

DM_DRAGOVER Sent to the window under the pointer as the pointer is dragged across it. A single DM_DRAGOVER message is sent each time the pointer moves and each time a key is pressed or released, and it contains a pointer to the DRAGINFO structure. The target can access this structure with the DrgAccessDraginfo function.

DM_DRAGLEAVE Sent whenever the DM_DRAGOVER message is sent to a window, and the pointer is moved outside the bounds of that window. If the target or an object in the window had been emphasized as a target, it should be de-emphasized.

Notes:

1. Container windows monitor the position of the pointer on DM_DRAGOVER messages and simulate the DM_DRAGLEAVE message when the pointer moves on or off a contained object.
2. A DM_DRAGLEAVE message is not sent if the user drops the objects being dragged within the window. Therefore, when DM_DROP is received, the application de-emphasizes any target that was emphasized as a valid target.

DM_DROP Sent to the target to provide it with the information necessary to establish a conversation for data exchange with the source. The target should immediately remove any target emphasis. The data transfers must not be done before responding to the DM_DROP message.

DM_DROPHELP Posted to a target to indicate that the user requested help for the drag operation while over that target.

Responding to Messages and Providing Visible Feedback

The DM_DRAGOVER message is sent to a target whenever the user drags the pointer into the window. To assess whether a drop can be accepted, the target must use the DrgAccessDraginfo function to get access to the DRAGINFO structure. It then determines whether a drop can be accepted for each object. Several factors are involved in making this determination, including the following:

- Both the source and target must support at least one common rendering mechanism and format.
- The target must understand at least one of the data types for the object.

Following are the four possible responses available to the target:

DOR_DROP The target should send DOR_DROP in response to the DM_DRAGOVER message if the objects being dragged are acceptable. The drop will not occur unless DOR_DROP is returned.

DOR_NODROP The target should send DOR_NODROP if the objects being dragged are acceptable, and the current operation is supported by the target; but the objects cannot be dropped in the current location in the target window. For example, a list box might return DOR_NODROP if it contains objects that can be dropped on, but the pointer is over an object that cannot be dropped on.

If the target response is DOR_NODROP, the DM_DRAGOVER message will continue to be sent to it when:

- The pointer is moved.
- A keyboard key is pressed.
- The pointer is moved out of and back into the window.

DOR_NODROPOP The target should send DOR_NODROPOP if it can accept the objects being dragged, but does not support the current operation. This response implies that the drop may be valid if the drag operation changes.

Once the target has sent DOR_NODROPOP, no further DM_DRAGOVER messages will be sent to it until:

- A keyboard key is pressed.
- The pointer is moved out of and back into the window.

DOR_NEVERDROP The target should use this response when it never will accept the objects being dragged. Once the target has responded with DOR_NEVERDROP to a DM_DRAGOVER message, no further DM_DRAGOVER messages will be sent to that target until the pointer is moved out of and back into the target window.

If a reply other than DOR_DROP is received from a target, the augmentation emphasis is automatically changed to indicate that no drop is allowed. This gives the user a visible cue that a drop cannot occur. The emphasis will revert to *drop allowed* when a DOR_DROP reply is received from some target.

Providing Customized Images

The target can provide a customized pointer to be displayed while it is the target of the drop by calling DrgSetDragPointer before responding to the DM_DRAGOVER message. It also can provide a customized image (icon, bit map, and so forth) to be displayed while it is the target by calling DrgSetDragImage. This capability may be used by a target to provide additional visible feedback to the user. The pointer will revert to the default when it is moved to a new target.

Providing Target Emphasis

The target should provide target emphasis so the user knows exactly where the drop will occur or, if the drop is not allowed, the boundaries of the region where the drop is not allowed.

If the user drags the pointer outside the target window, resulting in a new target, a DM_DRAGLEAVE message is sent to the former target. The receiver of a DM_DRAGLEAVE message should use it to de-emphasize the target, thus providing the user with visible feedback that this is no longer the target.

A container window should emphasize a target object by drawing a thin black rectangle around it. The application should use DrgGetPS and DrgReleasePS to obtain the presentation space in which to draw target emphasis.

Keyboard Augmentation

A direct manipulation operation begins in a *default state*, which means that when the user drops the object or objects on a target, the target is informed that it should perform its default operation. It is the target's responsibility to define its default operation. For a container window, the default should be a Move operation, if it is supported. The default for a device, such as a printer, should be a Copy operation.

As the user drags the object or objects, the default operation can be overridden by pressing and holding one of the following augmentation keys:

Ctrl Changes the operation to a Copy.

Shift Changes the operation to a Move.

Ctrl + Shift Changes the operation to a Link.

The last key pressed and held at the time of the drop determines the operation to be performed. The target can determine the defined augmentation key that was pressed at the time of the drop by inspecting the **usOperation** field of the DRAGINFO structure.

A target can define additional augmentation keys for its own use. In this case, **usOperation** would indicate that the operation is unknown, and the target needs to use the WinGetKeyState function to determine the actual augmentation key that was used.

Operation Emphasis: As the user presses augmentation keys, the pointer currently being displayed is modified to provide the user with a visible cue as to the type of operation being performed.

Summary of Functions Used by the Target

The following table summarizes the functions a target would use in direct manipulation:

Table 33-2 (Page 1 of 2). Summary of Functions Used by the Target

Function Name	Description
DrgAcceptDroppedFiles	Handles the file direct manipulation protocol for a given window.
DrgAccessDraginfo	Provides access to the shared segment containing the DRAGINFO structure.
DrgDeleteDraginfoStrHandles	Does a DrgDeleteStrHandle for all string handles in a DRAGINFO structure.
DrgDeleteStrHandle	Disassociates a string from the handle that was assigned to it by DrgAddStrHandle.
DrgDragFiles	Begins a direct manipulation operation for one or more files.
DrgFreeDraginfo	Releases the memory associated with a DRAGINFO structure. This function should be called when the target no longer needs the DRAGINFO structure or has previously called DrgAccessDraginfo, or a drop has occurred.
DrgFreeDragTransfer	Frees the storage associated with a DRAGTRANSFER structure.

Table 33-2 (Page 2 of 2). Summary of Functions Used by the Target	
Function Name	**Description**
DrgGetPS	Unlocks the screen and returns a handle to a cached presentation space that the target can use to provide target emphasis.
DrgPostTransferMsg	Posts a message to the other application involved in the direct manipulation.
DrgPushDragInfo	Gives a process access to a DRAGINFO structure.
DrgQueryDragItem	Copies a given object in a DRAGINFO structure.
DrgQueryDragItemCount	Returns the number of objects involved in a drag operation.
DrgQueryDragItemPtr	Returns a pointer to a given *DRAGITEM* structure.
DrgQueryNativeRMF	Returns the ordered pair representing the native rendering mechanism and format for an object.
DrgQueryNativeRMFLen	Returns the length of the string representing the native rendering mechanism and format of an object, excluding the null terminating byte.
DrgQueryStrName	Returns the contents of a string associated with a given string handle that was created by DrgAddStrHandle.
DrgQueryStrNameLen	Returns the length of the string associated with a given string handle that was created by DrgAddStrHandle.
DrgQueryTrueType	Returns the string representing the true type of an object being dragged.
DrgQueryTrueTypeLen	Returns the length of the string representing the true type of an object being dragged, *excluding* the null terminating byte.
DrgReleasePS	Releases the cache presentation space obtained using the DrgGetPS function.
DrgSendTransferMsg	Sends a message to the other application involved in the direct manipulation.
DrgSetDragImage	Enables a target to provide a customized image to be dragged.
DrgSetDragPointer	Enables a target to provide a customized image while it is the target of a drop.
DrgVerifyNativeRMF	Verifies that the native rendering mechanism and format for an object being dragged is one of a set of application-supplied rendering mechanisms and formats.
DrgVerifyRMF	Verifies that an application-specified rendering mechanism and format is valid for an object being dragged.
DrgVerifyTrueType	Verifies that an application-specified type is the true type of the object being dragged.
DrgVerifyType	Verifies that an application-specified type is valid for an object being dragged.
DrgVerifyTypeSet	Returns the intersection between the contents of the string represented by the type string handle and an application-supplied type string.

Two-Object Drag

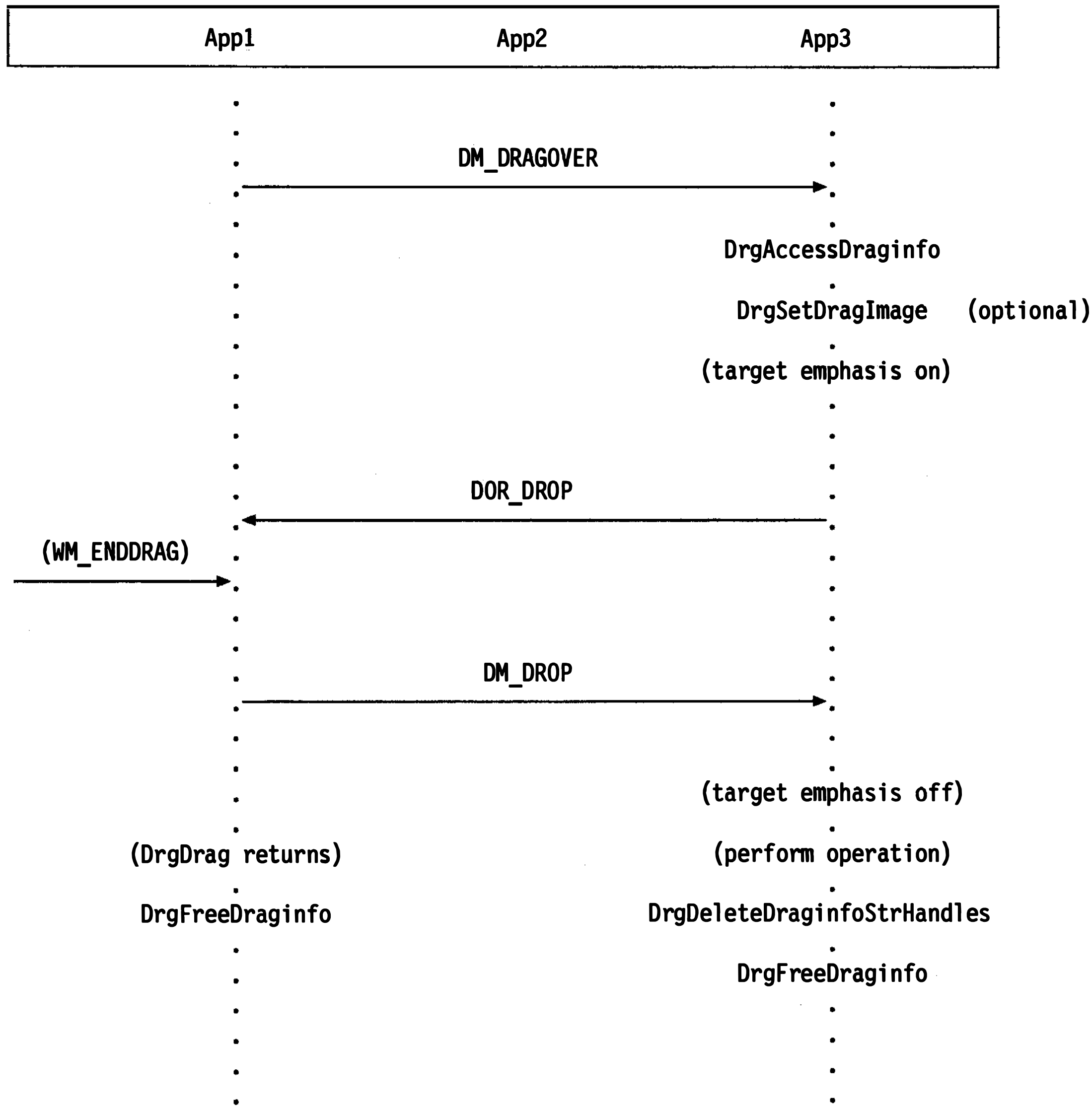
The following diagram represents the sequence of API functions and message flows for a typical direct manipulation operation. The flow shows a two-object drag from App1 to App3, dragging over App2. For this example, assume that App1 is implementing the Button 2 drag model.

Two-Object Drag

```
            App1                  App2                    App3

                     .                     .                       .
 (user select)       .                     .                       .
 ------------------->.                     .                       .
 (user select)       .                     .                       .
 ------------------->.                     .                       .
 WM_BEGINDRAG        .                     .                       .
 ------------------->.                     .                       .
                     .                     .                       .
                     .                     .                       .
         DrgAllocDraginfo                  .                       .
                     .                     .                       .
          DrgAddStrHandle                  .                       .
                     .                     .                       .
          DrgAddStrHandle                  .                       .
                     .                     .                       .
          DrgSetDragitem                   .                       .
                     .                     .                       .
          DrgSetDragitem                   .                       .
                     .                     .                       .
             DrgDrag                       .                       .
                     .                     .                       .
                     .    DM_DRAGOVER      .                       .
                     .-------------------->.                       .
                     .                     .                       .
                     .           DrgAccessDraginfo                 .
                     .                     .                       .
                     .            DrgSetDragImage   (optional)     .
                     .           (verify that drop                 .
                     .            can be accepted)                 .
                     .          (target emphasis on)               .
                     .            DrgFreeDraginfo                  .
                     .                     .                       .
                     .                     .                       .
                     .     DOR_DROP        .                       .
                     .<--------------------.                       .
                     .                     .                       .
                     .    DM_DRAGLEAVE     .                       .
                     .-------------------->.                       .
                     .                     .                       .
                     .         (target emphasis off)               .
                     .                     .                       .
                     .                     .                       .
```

Two-Object Drag (continued)



Application Interaction after a Drop

This portion of the document addresses aspects of a direct manipulation operation that need to be considered after a drop has occurred. See "Conversation after the Drop" on page 33-17 for an example.

Conversation Initiation

Direct manipulation offers various ways for both a source and target application to exchange data. To accomplish the exchange, a separate conversation must be established to transfer each data object from the source to the target. It is the responsibility of the target to inform the source about the rendering mechanism it wants to use and the format in which the data is to be exchanged. The target can establish the conversations to run in parallel, or it can initiate the conversations in a serial fashion. The following sections explain how each conversation is established.

Considerations when Establishing a Conversation

A source application may be able to exchange data with a target through several mechanisms, such as:

- Dynamic Data Exchange (DDE)
- OS/2 File
- Print.

Additionally, the source application might be able to render the data in various formats. For example, a spreadsheet application might be able to render its contents in a spreadsheet or text format. The ability of the source application to render the data in some format might, itself, depend on the exchange mechanism used. The rendering mechanisms and formats that a source application can support, for each object dropped, are provided to the target through the **hstrRMF** field in the DRAGITEM structure.

The first ordered pair in the set of rendering mechanisms and formats that the source application supports is the object's native rendering mechanism and format. This is the mechanism that most naturally conveys the data, either where it is now, or where it can be put most easily. The format conveys all information about the data. For example, a spreadsheet cell has a location in a row and column of a spreadsheet. Rendering the spreadsheet cell in a simple text format would cause this information to be lost, so a more appropriate format should be chosen for its native rendering format.

The target application also may be able to exchange data with the source through several different combinations of mechanism and format. It is the responsibility of the target to obtain the data from the source in the format that they both support and that provides the highest level of information about the data.

While making this determination, the target must consider the exchange capabilities offered by the mechanism. For example, an OS/2 File exchange mechanism can provide only a snapshot of the data at the time the direct manipulation operation occurred. An exchange using DDE, on the other hand, offers the target an opportunity to remain informed about changes to the data.

Determining Whether Data Can be Exchanged

During the drag portion of a drag-and-drop operation, the target must determine if it can exchange or receive data from the source for each object involved in the operation. The object must meet the following minimum requirements to exchange data:

- The source and target must share knowledge of at least one common type for the object. The target can make this determination by using the DrgVerifyTypeSet or DrgVerifyType function.
- The source and target must share at least one common rendering mechanism and format for that type object. The target can make this determination by using the DrgVerifyRMF function.

When these conditions are met, a target can let the object be dropped.

Determining How To Exchange the Data

The target determines which rendering mechanism and format to use in the following manner:

1. Uses the native rendering mechanism and format whenever possible.

   This rendering conveys *ALL* information about the data. A target can determine if it supports the native rendering mechanism and format through the use of the following functions:

   - DrgVerifyNativeRMF
   - DrgQueryNativeRMFLen
   - DrgQueryNativeRMF

   Regardless of whether the native rendering mechanism and format supported by the source can be used, the target can elect to exchange the data in a rendering mechanism and format that conveys less information about the object.

2. Uses the next best rendering mechanism and format.

   This is especially good for a Copy operation, because the user does not lose data about the object as occurs when the object is moved.

   The target can determine the next best rendering mechanism and format to use through repeated calls to the DrgVerifyRMF function. The calls are made starting with the most desirable rendering mechanism and format pair and progressing to the least desirable pair. Once a pair that the source supports has been found, the target can exchange the data.

Performance Considerations

When context information about an object will be lost because of using a less-desirable rendering mechanism and format, the target can elect to pick a common mechanism and format that will achieve the best performance. This is done the same way the next best rendering mechanism and format is selected, proceeding from the best-performing rendering to the worst.

Using Direct Manipulation Data Transfer in an Application

Some standard rendering mechanisms are already defined but this system lets the set of rendering mechanisms be expanded, allowing for:

- Additional standard rendering mechanisms to be defined in the future
- Application definition of private or nonstandard rendering mechanisms.

An application can elect to support some, all, or none of the standard rendering mechanisms defined by the system. Applications that do not support any of the standard rendering mechanisms are not precluded from using direct manipulation. However, support of the standard rendering mechanisms and formats increases the chances of a successful data transfer between applications.

An application that supports a particular rendering mechanism, whether or not it is a rendering mechanism defined by the system, must follow a specific set of guidelines defined by that rendering mechanism, including conversation-initiation procedures and naming conventions. The guidelines for the current system-defined rendering mechanisms are described in the following sections.

Regardless of the rendering mechanism used, it may be necessary to prepare the source for the rendering of the object. Such an action is necessary when a window needs to be created by the source in order to handle the conversation. This is done by sending a DM_RENDERPREPARE message to the **hwndSource** window in the DRAGINFO structure. This message need be sent only when the DC_PREPARE flag is on in the **fsControl** field of the DRAGITEM structure. When the source receives this message, it performs any necessary preparation for the rendering and fills in the **hwndItem** field in the DRAGITEM structure, allowing the target to establish conversation with that window.

Conversation after the Drop

The following diagram represents the sequence of message flows for a typical direct manipulation data-transfer operation. The flow describes a single-object move from source to target. The user dropped on white space in the target container.

For this example, assume that the rendering mechanism selected is DRM_OS2FILE and that the source does not initially provide the target with the source item's file name. Also assume that the source and target items exist on different drives.

Source	Target

```
                 .                                  .
                 .                                  .
                 .                        DrgAllocDragtransfer
                 .                                  .
                 .                         DrgSendTransferMsg
                 .                                  .
                 .            DM_RENDER             .
                 .<---------------------------------.
                 .                                  .
 Verify the rendering mechanism                     .
           and format                               .
                 .                                  .
          DrgAddStrHandle                           .
          (hstrSourceName)                          .
                 .                                  .
                 .         DM_RENDER(reply)         .
                 .--------------------------------->.
                 .                                  .
              DosCopy                               .
                 .                                  .
        DrgFreeDragtransfer                         .
                 .                                  .
                 .        DM_RENDERCOMPLETE         .
                 .--------------------------------->.
                 .                                  .
                 .        DM_ENDCONVERSATION        .
                 .<---------------------------------.
                 .                                  .
             DosDelete                    DrgFreeDragtransfer
                 .                                  .
          DrgFreeDraginfo                  DrgDeleteStrHandle
                 .                                  .
                 .                           DrgFreeDraginfo
                 .                                  .
                 .                                  .
```

Standard Rendering Mechanisms

The following sections describe the standard rendering mechanisms used by various containers and applications for direct manipulation.

OS/2 File Rendering Mechanism

This rendering mechanism can be used by various containers, including file folders and trash cans. These containers allow objects to be dragged and dropped on white space in the container to accomplish a Move or Copy operation. They also can allow objects in the same or another container to be dragged and dropped on objects within the container to accomplish an operation.

Mechanism Name: The string for this rendering mechanism is DRM_OS2FILE.

Messages: The following messages are used by the DRM_OS2FILE:

- DM_RENDER

  This message is sent by a target to a source to request a rendering for an object. When this message is received, the source determines if it understands the rendering mechanism and format selected by the target for the object. It also confirms that it allows the operation selected by the user for that object. The source must respond to this message before proceeding with the rendering operation.

- DM_RENDERCOMPLETE

  This message is posted by a source to a target to notify the target that the rendering operation has been completed by the source, either successfully or unsuccessfully. The source can elect to let the target retry a successful or an unsuccessful operation. In this case, it should return to its state at the time of the drop for that object and indicate, in the message, that a retry is allowed.

  Support for this message by a source is optional. If this message is not supported, then:

  - The source must convey all necessary information to the target order to allow it to handle the rendering operation.
  - It must always indicate that native rendering is allowed when replying to a DM_RENDER message.

- DM_ENDCONVERSATION

  This message is sent by a target to a source to notify the source that the rendering operation is complete and that the conversation is terminated. When this message is received, the entire drop operation for the object is complete. The source can now release any resources it had allocated to the drop and rendering operations. When the reply is received, the target can release the resources it had allocated to the operation.

Native Rendering by the Target: If the target understands the native rendering mechanism and format of the object, it may be possible to render the object without any involvement on the part of the source, provided the source has given the target sufficient information to do so. In order for the rendering to be performed by the target, the source must fill in, at a minimum, the **hstrContainerName** and **hstrSourceName** fields. This **hstrContainerName** field represents the subdirectory that the file indicated by **hstrSourceName** is in. For the target to do the rendering on its own, the true type of the object must be DTYP_OS2FILE. When these conditions are met, the target may proceed with the operation. When the operation is

complete, the target must send a DM_ENDCONVERSATION message to the window indicated by **hwndItem** in the DRAGITEM structure.

Preventing a Target from Rendering an Item: A source can prevent a target from doing the rendering operation on its own by not providing the source name for the object. This may be a necessary action for sources that implement some type of security, or that may not allow particular operations to be performed for an object move. When a source takes this course, it must fill in the **hstrSourceName** in the DRAGITEM structure before replying to a DM_RENDER message. The target will delete the **hstrSourceName** string handle prior to freeing the DRAGINFO structure, just as it would if the information had been passed to it at the time of the drop.

Requesting the Source to Render the Item: Whenever the conditions for a target to do the rendering operation without source participation are not met, the target must request the source to carry out the rendering by posting a DM_RENDER message to the source. Of course, the target can do this even if it is able to carry out the rendering mechanism on its own.

Allocating and Freeing a DRAGTRANSFER Structure: The data in a drag transfer message is carried in a DRAGTRANSFER structure. DRAGTRANSFER structures are allocated when the target calls DrgAllocDragtransfer.

When the conversation or conversations are completed, both the source and the target must call DrgFreeDragtransfer to free the shared memory. The target should do it immediately after sending a DM_ENDCONVERSATION message. The source should do it immediately after sending a DM_RENDERCOMPLETE message.

Operation Specifics: Regardless of the operation being performed, the target must fill in the **hstrRenderToName** field in the DRAGTRANSFER structure before sending a DM_RENDER message. This is the fully qualified drive, path, and file name of the file that will contain the data when the rendering operation is complete. When the source has completed the operation, it must post a DM_RENDERCOMPLETE message to the target. The target then must complete the direct manipulation operation for that object by posting a DM_ENDCONVERSATION message to the source. Once the conversations for all of the objects involved in the drop are complete, the target can delete the string handles and free the DRAGINFO structure.

Non-Native Mechanism Actions: The target may select the DRM_OS2FILE rendering mechanism when it is not the native rendering mechanism for an object, as long as the source supports it. In this case, the target must always request that the source carry out the rendering operation as described above. The source should render the data in the requested format to the file specified by the **hstrRenderToName** field. If the requested operation is a Move, the source should take whatever action is necessary to remove its knowledge of the object as long as no information regarding the object was lost in the transfer.

Naming Conventions: The naming conventions for this rendering mechanism follow:

- hstrContainerName

  Contains the fully qualified drive and path name for the source file.

  Examples are:

  ```
  C:\
  C:\MYSUBDIR\
  A:\SUBDIR1\SUBDIR2\
  \\NETWORK\SHARED\SUBDIRA\SUBDIRB\
  ```

- hstrSourceName

  Contains the name of the source file or subdirectory, for example:

  ```
  MYSOURCE.C
  MYSOURCE.H
  MYSOURCE IS A LONG FILE NAME
  SUBDIR3
  ```

- hstrRenderToName

  Contains the fully qualified file or subdirectory name that is to be used at the target, for example:

  ```
  C:\MYSUBDIR\MYSOURCE.C
  \\NETWORK\SHARED\SUBDIRA\SUBDIRB\MYSOURCE.H
  C:\SUBDIR1\SUBDIR2\SUBDIR3
  ```

Types: Any type that is allowed as a *.TYPE* extended attribute is allowed in the **hstrType** field of the DRAGITEM structure. The type for a file may be obtained using the DosQFileInfo function, and set by using the DosSetFileInfo function.

Print Rendering Mechanism

A common object that might be provided by a container is a printer. This object would allow objects to be dragged and dropped on it to accomplish a print operation.

Mechanism Name: The string for this rendering mechanism is DRM_PRINT.

Messages: To support this rendering mechanism, a source must be able to receive and process a DM_PRINT message. The target will post this message to the source. When the message is received, the source prints the current view of the object identified in the message to the printer queue, which is also identified in the message.

Native Mechanism Actions: There are no native mechanism actions for this rendering mechanism, because the act of printing an object is considered a transform from the native rendering mechanism to the print mechanism.

Naming Conventions: None.

Dynamic Data Exchange (DDE) Rendering Mechanism

This rendering mechanism can be used by various containers and applications. The containers allow objects to be dragged and dropped on white space in the container to accomplish a Move or Copy operation. They also can allow objects in the same or another container to be dragged and dropped on objects within the container to accomplish some operation.

Mechanism Name: The string for this rendering mechanism is DRM_DDE.

Messages: To support this rendering mechanism, a source must be able to receive and process the following messages:

- WM_DDE_REQUEST

  This message is posted by the target to the window indicated by the **hwndItem** field in the DRAGITEM structure to request information regarding the object. Note that WM_DDE_INITIATE is not required because the target already has the handle of the window it wants to converse with. This message is sent for all Move and Copy operations.

- WM_DDE_ADVISE

  This message is posted by the target to the window indicated by the **hwndItem** field in the DRAGITEM structure order to maintain a *hot link* to the object.

- WM_DDE_UNADVISE

  This message is posted by the target to the window indicated by the **hwndItem** field in the DRAGITEM structure to terminate a hot link to the object.

- WM_DDE_TERMINATE

  This message is posted by the target to the window indicated by the **hwndItem** field in the DRAGITEM structure to terminate a conversation.

To support this rendering mechanism, a target must be able to receive and process the following messages:

- WM_DDE_DATA

  This message is posted to the target by the source to deliver the requested information regarding the object.

- WM_DDE_ACK

  This message is posted to the target by the source to acknowledge a WM_DDE_ADVISE or WM_DDE_UNADVISE message.

- WM_DDE_TERMINATE

  This message is posted to the target by the source to end a conversation.

Native Mechanism Actions: Prior to establishing a DDE conversation, the target should determine the source-supported formats in which it wants to have the object rendered. It should register this format in the system atom table, and use the resulting atom in the **usFormat** field of the DDESTRUCT used in the conversation.

The target should establish the DDE conversation by posting a WM_DDE_REQUEST message to the window indicated by the **hwndItem** field in the DRAGITEM structure. The target acts as the client, and the source acts as the server in the conversation.

Operation Specifics: The following actions should be taken by the source, depending on the operation being performed:

Copy Send the data to the target.

Move Remove knowledge of the object after receiving confirmation that the target has successfully completed its portion of the rendering operation.

Non-Native Mechanism Actions: The target and source proceed in the same way, regardless of whether DDE was the native rendering mechanism or an alternate rendering mechanism.

Naming Conventions: The naming conventions for the DRM_DDE rendering mechanism follow:

- hstrSourceName

  Contains the object name to be used in the DDE conversation.

- hstrRMF

  The format portion of the list of ordered pairs in the format *<DRM_DDE,format>* identifies the formats supported by the source for the object. The non-standard

DDE formats that these formats map to must be registered in the system atom table by both the source and the target.

Types: Any type that is allowed as a *.TYPE* extended attribute is allowed in the **hstrType** field of the DRAGITEM structure.

Application Extensions to the Direct Manipulation Data Transfer Protocol

An application can choose to define a new rendering mechanism. However, if an application intends to provide renderings from this extended rendering mechanism to existing rendering mechanisms, it should publish enough information so that other application developers can use the new mechanism. An application must address several distinct areas of definition. These areas are described below, in general, and also are addressed under the definition for the system mechanisms.

Rendering Mechanism Name

The string name of the rendering mechanism should be defined by the application. This string name will be specified in the mechanism/format pair of the DRAGITEM structure.

Native Mechanism Actions

When both a source and target application store the data in the same native mechanism, a transform is not required. Instead, the native Move and Copy actions for that mechanism could be performed by the target. An application must completely define the proper procedure for performing that action. In the case of files, the native Move action is defined as a DosMove or DosCopy/DosDelete. The native Copy action is DosCopy. An application need not support all of the basic actions; it can choose to define additional native mechanism actions, indicated by the DO_UNKNOWN action in the DRAGINFO structure.

Naming Conventions

An application that is defining a new mechanism must completely specify the naming conventions for objects rendered in that mechanism. This information typically includes both the name of the data and preceding information describing the exact location of the data. Any special rules concerning uppercase and lowercase or character sets to be used in naming also must be specified. The semantics for using these mechanism names, as well as an algorithm for deriving location information, also must be defined.

An application that is defining a new rendering mechanism must completely define the set of messages that a target and source application must support, and specify the appropriate action to be taken for each message. The message IDs (above WM_USER) for the messages must be published.

Performance Considerations

If an application provides or defines transforms from the newly defined mechanism to existing mechanisms, performance information about the transform between mechanisms should be provided. This will aid the application developer in choosing the appropriate transform when it encounters an application that transforms from an unknown native mechanism to several different known mechanisms.

Summary

The following tables describe the structures and messages used in direct manipulation:

Table 33-3. Direct Manipulation Structures

Structure name	Description
DRAGIMAGE	Dragged-image structure.
DRAGINFO	Drag-information structure.
DRAGITEM	Drag-object structure.
DRAGTRANSFER	Drag-conversation structure.

Table 33-4 (Page 1 of 2). Direct Manipulation (Drag) Messages

Message	Description
DM_DRAGERROR	Sent to the caller of DrgDragFiles or DrgAcceptDroppedFiles when an error occurs during a move or copy operation.
DM_DRAGFILECOMPLETE	Sent when a direct manipulation operation on a file is complete.
DM_DRAGLEAVE	Sent to a window that is being dragged over when one of the following occurs: • The object is dragged outside the boundaries of the window. • The drag operation is terminated while the object is over the window.
DM_DRAGOVER	Lets the window under the pointer determine whether the object currently being dragged can be dropped.
DM_DRAGOVERNOTIFY	Sent to the source of a drag immediately after a DM_DRAGOVER message is sent to a target window.
DM_DROP	Sent to the target when the dragged object is dropped.
DM_DROPHELP	Requests help for the current drag operation.
DM_EMPHASIZETARGET	Sent to the caller of DrgAcceptDroppedFiles to tell it to either apply or remove target emphasis from itself.
DM_ENDCONVERSATION	The target used this message to notify a source that a drag operation is complete.
DM_FILERENDERED	Sent to the window handling the drag conversation for the caller of DrgDragFiles.
DM_PRINT	Sent to a source to request it to print the current view of an object.
DM_RENDER	Used to request a source to provide a rendering of an object in a specified rendering mechanism and format.
DM_RENDERCOMPLETE	Posted by a source to a target window.
DM_RENDERFILE	Sent to the caller of DrgDragFiles to tell it to render a file.

Table 33-4 (Page 2 of 2). Direct Manipulation (Drag) Messages	
Message	**Description**
DM_RENDERPREPARE	Tells a source to prepare for the rendering of an object.

Chapter 34. Window Timers

A *window timer* enables an application to post timer messages at specified intervals. This chapter describes how to use window timers in PM applications.

About Window Timers

A window timer causes the system to post WM_TIMER messages to a message queue at specified time intervals called *timeout values*. A timeout value is expressed in milliseconds.

An application starts the timer for a given window, specifying the timeout value. The system counts down approximately that number of milliseconds and posts a WM_TIMER message to the message queue for the corresponding window. The system repeats the countdown-post cycle continuously until the application stops the timer.

The timeout value can be any value in the range from 0 through 65535. However, the operating system cannot guarantee that all values are accurate. The actual timeout depends on how often the application retrieves messages from the queue and the system clock rate. In many computers, the operating system clock ticks about every 50 milliseconds, but this can vary widely from computer to computer. In general, a timer message cannot be posted more frequently than every system clock tick. To make the system post a timer message as often as possible, an application can set the timeout value to 0.

An application starts a timer by using the WinStartTimer function. If a window handle is given, the timer is created for that window. In such case, the WinDispatchMsg function dispatches the WM_TIMER message to the given window when the message is retrieved from the message queue. If a NULL window handle is given, it is up to the application to check the message queue for WM_TIMER messages and dispatch them to the appropriate window.

A new timer starts counting down as soon as it is created. An application can reset or change a timer's timeout value in subsequent calls to the WinStartTimer function. To stop a timer, an application can use the WinStopTimer function.

The system contains a limited number of timers that must be shared among all PM applications; each application should use as few timers as possible. An application can determine how many timers currently are available by checking the CV_TIMERS system value.

Every timer has a unique timer identifier. An application can request that a timer be created with a particular identifier or have the system choose a unique value. When a WM_TIMER message is received, the timer identifier is contained in the first message parameter. Timer identifiers enable an application to determine the source of the WM_TIMER message.

Three timer identifiers are reserved by and for the system and cannot be used by applications; these system timer identifiers and their symbolic constants are shown in the following table:

Table 34-1. System Timers

Value	Meaning
TID_CURSOR	Identifies the timer that controls cursor blinking. Its timeout value is stored in the os2.ini file under the CursorBlinkRate keyname in the PM_ControlPanel section.
TID_FLASHWINDOW	Identifies the window-flashing timer.
TID_SCROLL	Identifies the scroll-bar repetition timer that controls scroll-bar response when the mouse button or a key is held down. Its timeout value is specified by the system value SV_SCROLLRATE.

WM_TIMER messages, like WM_PAINT and semaphore messages, are not actually posted to a message queue. Instead, when the time elapses, the system sets a record in the queue indicating which timer message was posted. The system builds the WM_TIMER message when the application retrieves the message from the queue.

Although a timer message may be in the queue, if there are any messages with higher priority in the queue, the application retrieves those messages first. If the time elapses again before the message is retrieved, the system does not create a separate record for this timer, meaning that the application should not depend on the timer messages being processed at precise intervals. To check the accuracy of the message, an application can retrieve the actual system time by using the WinGetCurrentTime function. Comparing the actual time with the time of the previous timer message is useful in determining what action to take for the timer.

Using Window Timers

There are two methods of using window timers. In the first method, you start the timer by using the WinStartTimer function, supplying the window handle and timer identifier. The function associates the timer with the specified window. The following code fragment starts two timers: the first timer is set for every half second (500 milliseconds); the second, for every two seconds (2000 milliseconds).

```
WinStartTimer(hab, /* Anchor-block handle */
    hwnd,          /* Window handle       */
    ID_TIMER1,     /* Timer identifier    */
    500);          /* 500 milliseconds    */

WinStartTimer(hab, /* Anchor-block handle */
    hwnd,          /* Window handle       */
    ID_TIMER2,     /* Timer identifier    */
    2000);         /* 2000 milliseconds   */
```

Once these timers are started, the WinDispatchMsg function dispatches WM_TIMER messages to the appropriate window. To process these messages, add a WM_TIMER case to the window procedure for the given window. By checking the first parameter of the WM_TIMER message, you can identify a particular timer, then carry out the actions related to it. The following code fragment shows how to process WM_TIMER messages:

```
case WM_TIMER:
    switch (SHORT1FROMMP(mp1)) { /* Obtains timer identifier */
        case ID_TIMER1:
            .
            . /* Carry out timer-related tasks.            */
            .
            return 0;

        case ID_TIMER2:
            .
            . /* Carry out timer-related tasks.            */
            .
            return 0;
    }
```

In the second method of using a timer, you specify NULL as the *hwnd* parameter of the WinStartTimer call. The system starts a timer that has no associated window and assigns an arbitrary timer identifier. The following code fragment starts two window timers using this method:

```
ULONG idTimer1, idTimer2;

idTimer1 = WinStartTimer(hab, (HWND) NULL, 0, 500);
idTimer2 = WinStartTimer(hab, (HWND) NULL, 0, 2000);
```

These timers have no associated window, so the application must check the message queue for WM_TIMER messages and dispatch them to the appropriate window procedure. The following code fragment shows a message loop that handles the window timers:

```
HWND hwndTimerHandler; /* Handle of window for timer messages */
QMSG qmsg;             /* Queue-message structure             */

while (WinGetMsg(hab, &qmsg, (HWND) NULL, 0, 0)) {
    if (qmsg.msg == WM_TIMER)
        qmsg.hwnd = hwndTimerHandler;
    WinDispatchMsg(hab, &qmsg);
}
```

You can use the WinStopTimer function at any time to stop a timer. The following code fragment demonstrates how to stop a timer:

```
WinStopTimer(hab, hwnd, ID_TIMER1);      /* Stops first timer */
```

Summary

Following are the OS/2 functions and the message used with window timers:

Table 34-2. Window Timer Functions

Function Name	Description
WinGetCurrentTime	Returns the current time.
WinStartTimer	Starts a timer.
WinStopTimer	Stops a timer.

Table 34-3. Window Timer Message

Message	Description
WM_TIMER	Posted when a timer times out.

Chapter 35. Atom Tables

Atom tables enable applications to generate unique identifiers and manage strings. This chapter describes how to use atom tables in PM applications.

About Atom Tables

An *atom table* is an operating system mechanism that an application uses to obtain identifiers that are unique, system-wide, and to manage strings efficiently. An application places a string, called an *atom name*, in an atom table and receives a 32-bit integer value, called an *atom*, that the application can use to access that string.

System Atom Table

The *system atom table* is available to all applications. When an application places a string in the system atom table, any application that has the atom name can obtain the atom by querying the system atom table.

An application that defines messages, clipboard-data formats, or dynamic data exchange (DDE) data formats that are intended for use among applications must place the names of the messages or formats in the system atom table. So doing avoids possible conflicts with messages or formats defined by the system or other applications, and makes the atoms for the messages or formats available to other applications. Applications should use names that are not likely to be used by other applications for other purposes.

Some PM functions enable applications to use atoms in parameters that normally take pointers to strings. For example, the WinRegisterClass function takes a pointer to a string for its *pszClassName* parameter. WinRegisterClass places the class name string in the system atom table. Afterward, an application can query the system atom table to obtain the atom, then use the atom as the *pszClientClass* parameter of the WinCreateStdWindow function. This process can save space in the data segment of applications that create many windows of the same private class.

Private Atom Tables

An application can use a *private atom table* to efficiently manage a large number of strings that are used only within the application. The strings in a private atom table, and the resulting atoms, are available only to the application that created the table.

An application that must use the same string in a number of data structures can save data-segment space by using a private atom table. Rather than copying the string into each data structure, the application can place the string in the atom table and use the resultant atom in the data structures. In this way, a string that appears only once in the data segment still can be used many times in the application.

Applications also can use private atom tables to save time when searching for a particular string. To perform a search, an application must place the search string in the atom table only once, then compare the resultant atom with the atoms in the relevant data structures. This usually is faster than doing string comparisons.

Atom-Table Handle

Every atom table has a unique handle. An application must obtain the handle before performing any atom operations. To obtain the handle of the system atom table, an application must use the WinQuerySystemAtomTable function. To create a private atom table and obtain its handle, an application must use the WinCreateAtomTable function. The atom-table handle returned by either of these calls must be used for all other atom functions.

An application that no longer needs its private atom table should call the WinDestroyAtomTable function to destroy the table and free the memory that the system allocated for the table.

Atom Types

Applications can use two *types* of atoms: string and integer.

String Atoms

Applications pass null-terminated strings to atom tables and receive *string atoms* (32-bit integers) in return. String atoms have the following properties:

- The maximum number of string atoms allowed is 16K. The values of string atoms are from *0xC000* through *0xFFFF*.
- The maximum amount of data that an atom table can store is 64K. This includes the control data that the operating system uses to manage the atom table (32 bytes for the table plus 6 bytes for each string atom).
- The maximum length of an atom name is 255 characters. A zero-length string is not a valid atom name.
- Case is significant when searching for an atom name in an atom table, and the entire string must match. No substring matching is performed.
- A usage count is associated with each atom name. The count is incremented each time the atom name is added to the table and decremented each time the atom name is deleted from the table. This allows different users of the same string atom to avoid destroying each other's atom names. When the usage count for an atom name equals zero, the system removes the atom and atom name from the table.

Integer Atoms

Integer atoms differ from string atoms as follows:

- Integer atoms are values from *0x0001* through *0xBFFF*. The values of integer atoms and string atoms do not overlap, so the two types of atoms can be intermixed.
- The string representation of an integer atom is *ddddd*, where *ddddd* are decimal digits. Leading zeros are ignored.
- There is no usage count nor storage overhead associated with an integer atom.

The operating system uses integer atoms to detect whether the same window class name is being defined more than once. The system defines the predefined window class names using integer atoms as constants. When an application registers a window class, the system enters the specified class name in the system atom table. The system then compares the resultant atom with the predefined window-class constants and with the atoms representing the application-defined class names registered earlier. To be able to do this comparison, the system must express the preregistered class names as atoms. By defining the class names as integer atoms, the system ensures that the atoms do not conflict with the string atoms it generates for application-defined class names.

Atom Creation and Usage Count

An application creates an atom by calling the WinAddAtom function, passing an atom-table handle and a pointer to a string. The system searches the specified atom table for the string. If the string already resides in the atom table, the system increments the usage count for the string and returns the corresponding atom to the application. Repeated calls to add the same atom string return the same atom. If the atom string does not exist in the table when WinAddAtom is called, the string is added to the table, its usage count is set to 1, and a new atom is returned.

An application can retrieve the usage count associated with a given atom using the WinQueryAtomUsage function. By obtaining the usage count, an application can detect whether other applications, or other threads within the application, are using the same atom.

An application calls the WinDeleteAtom function when it no longer needs to use an atom. WinDeleteAtom reduces the usage count of the corresponding atom by 1. When the usage count reaches zero, the system deletes the atom name from the table.

Atom-Table Queries

An application can find out if a particular string is already in an atom table by using the WinFindAtom function. WinFindAtom searches the atom table for the specified string and, if the string is there, returns the corresponding atom.

There are two functions that an application can use to retrieve a string from an atom table, provided that the application has the atom corresponding to the desired string. The first, WinQueryAtomLength, returns the length of the string corresponding to the atom. This allows the application to create a buffer of the appropriate size for the string. An application uses the WinQueryAtomName function to retrieve the string and copy it to the buffer.

Atom String Formats

The second parameter to the WinAddAtom and WinFindAtom functions, *pszAtomName,* is a pointer to zero-terminated string. An application can specify this pointer in one of the following four ways:

Table 35-1. Atom String Formats

Format	Description
"!",atom	Points to a string in which the atom is passed indirectly, as a value.
#ddddd	Points to an integer atom specified as a decimal string.
long word: FFFF(low word)	Passes an atom directly. The atom is in the low word of the *pszAtomName* parameter. The operating system uses this format to add predefined window classes to the system atom table.
string atom name	The pointer is to a string atom name. Applications typically use this format to add an atom string to an atom table and receive an atom in return.

The *"!",atom* and *long word: FFFF(low word)* formats are useful when incrementing the usage count of an existing atom for which the original atom string is not known. For example, the system clipboard manager uses the *long word: FFFF(low word)* format to increment the usage count of each clipboard-format atom when that format is placed on the clipboard. By using this format, the atom is not destroyed even if the original user of the atom deletes the it, because the usage count still shows that the clipboard is using the atom.

Using Atom Tables

This section explains how to create unique window-message atoms, dynamic data exchange (DDE) formats and a clipboard format.

Creating Unique Window-Message Atoms

You must create atoms for your application-defined window messages if other applications are likely to recognize those messages. For example, your application might communicate with another application by using an agreed-upon message that is not defined by the system. Both applications must use the same string identifier for the shared message type—for example, OUR_LINK_MESSAGE. Each time the applications run, they add this string to the system atom table and receive an atom in return. Both applications register the same string in the system atom table, so they both receive the same atom. Then, this atom can be used to identify the message without conflicting with other system-wide message identifiers.

A consequence of using atoms to identify a window message is that the message cannot be decoded as a C-language case statement, as is usually done, because the value of the atom cannot be known until run time. Instead, you must add a default case that checks the value of the message against the value of the atoms you have registered.

The following code fragment shows how to add an application-defined message string to the system atom table, then use the resultant atom to broadcast and receive the message.

```
#define IDM_BROADCAST 25

HATOMTBL hatomtblSystem;
ATOM atomLinkMessage;
UCHAR szLinkMessage[] = "OUR_LINK_MESSAGE";

MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2)
{

    switch (msg) {
        case WM_CREATE:
            hatomtblSystem = WinQuerySystemAtomTable();
            atomLinkMessage = WinAddAtom(hatomtblSystem, szLinkMessage);
            return FALSE;

        case WM_COMMAND:
            if (SHORT1FROMMP(mp1) == IDM_BROADCAST) {
                WinBroadcastMsg(HWND_DESKTOP, atomLinkMessage,
                    (MPARAM) NULL, (MPARAM) NULL,
                    BMSG_DESCENDANTS | BMSG_POSTQUEUE);
            }
            return 0;

        default:

            /*
             * Check for the atom representing "OUR_LINK_MESSAGE".
             */

            if (msg == atomLinkMessage)
                return DoOurMessage(...);
            break;
    }
    return WinDefWindowProc(hwnd, msg, mp1, mp2);
}
```

Creating DDE Formats and a Unique Clipboard Format

Applications that define their own clipboard or DDE formats must register those formats in the system atom table to avoid conflicting with the predefined formats and any formats used by other applications.

The following code fragment registers a custom format:

```
#define MAX_BUF_SIZE 128

HAB hab;
HATOMTBL hatomtblSystem;
ATOM atomFormatID;
PSZ  pszSrc, pszDest;
BOOL fSuccess;
CHAR szClipString[MAX_BUF_SIZE];

/*
 * Get the handle of the system atom table, then add the format
 * name to the table.
 */

hatomtblSystem = WinQuerySystemAtomTable(); /* Sys. atom table handle */
atomFormatID = WinAddAtom(hatomtblSystem,   /* Register format string */
     "SuperCAD_FORMAT");
    .
    . /* Obtain data and write data to buffer (szClipString)         */
    .

if (WinOpenClipbrd(hab)) {        /* Open the clipboard          */

    /* Allocate a shared memory object for the text data.       */

    if (!(fSuccess = DosAllocSharedMem(
            (PVOID)&pszDest,       /* Pointer to shared memory object */
            (PSZ) NULL,            /* Use unnamed shared memory       */
            (ULONG)strlen(szClipString) + 1, /* Amount of memory      */
            PAG_WRITE  |           /* Allow write access              */
            PAG_COMMIT |           /* Commit the shared memory        */
            OBJ_GIVEABLE))) {      /* Make pointer giveable           */

        /* Set up the source pointer to point to text.          */
        pszSrc = szClipString;

        /* Copy the string to the allocated memory.             */
        while (*pszDest++ = *pszSrc++);

        /* Clear old data from the clipboard.                   */
        WinEmptyClipbrd(hab);

        /*
         * Pass the pointer to the clipboard in custom format. Notice
         * that the pointer must be a ULONG value.
         */

        fSuccess = WinSetClipbrdData(hab, /* Anchor block handle      */
            (ULONG) pszDest,              /* Pointer to text data     */
            atomFormatID,                 /* Custom format ID (atom)  */
            CFI_POINTER);                 /* Passing a pointer        */

        /* Close the clipboard. */
        WinCloseClipbrd(hab);
    }
}
```

Summary

The following OS/2 functions are associated with atom tables:

Table 35-2. Atom Table Functions

Function Name	Description
WinAddAtom	Adds an atom to an atom table.
WinCreateAtomTable	Creates an empty atom table of the specified size.
WinDeleteAtom	Deletes an atom from an atom table.
WinDestroyAtomTable	Destroys an atom table.
WinFindAtom	Find an atom in the atom table.
WinQueryAtomLength	Queries the length of an atom represented by the specified atom.
WinQueryAtomUsage	Returns the number of times an atom has been used.
WinQuerySystemAtomTable	Returns the handle of the system atom table.

Chapter 36. Initialization Files

Initialization files enable an application to store and retrieve information that the application uses when it starts up. This chapter describes how to use the OS/2 2.0 Profile Manager to create, manage, and use the system's initialization files. The following topics are related to this chapter:

- File system
- Presentation Manager interface applications.

About Initialization Files

An initialization file is a convenient place to store information between sessions. Profile Manager enables applications to create their own initialization files and to access the OS/2 initialization files, *os2.ini* and *os2sys.ini*. Just as the system uses the os2.ini and os2sys.ini files to store configuration information for system startup, an application can create an initialization file that stores information it uses to initialize windows and data.

The system initialization files contain sections and settings used by the PM applications (such as Desktop Manager, Control Panel, and Print Manager). Although applications can read settings from the initialization files, only rarely does an application need to change a setting. OS/2 initialization files are binary; the user cannot view or edit them directly.

An initialization file consists of one or more sections; each section contains one or more settings, or keys. Each key consists of two parts: a name and a value. Both section names and key names are null-terminated strings. The value assigned to a key can be a null-terminated string, a null-terminated string representing a signed integer, or individual bytes of data.

Once an initialization file is created, an application can rename, copy, move, or delete that file just as it does any other file. Although an application also could read directly to or write directly to the initialization file, the application should always use Profile Manager functions to access the contents of the file. Both character-based OS/2 applications and PM applications can use Profile Manager functions. Before calling Profile Manager, a thread must initialize an anchor block by using the WinInitialize function.

Using Initialization Files

This section explains how to use Profile Manager functions to perform the following tasks:

- Create, open, and close initialization files.
- Read and write settings.
- Identify the initialization files.

Creating, Opening, and Closing Initialization Files

You can create an initialization file or open an existing initialization file by using the PrfOpenProfile function. The function requires a handle to an anchor block and a pointer to the name of an initialization file. If the file does not exist in the given path, the function automatically creates an initialization file.

The following code fragment creates an initialization file named *pmtools.ini* in the current directory:

```
HAB hab;
HINI hini;

hab = WinInitialize(0);
if ((hini = PrfOpenProfile(hab, "pmtools.ini")) == NULL){
                     .
                     . /* File was not created */
                     .
   }
```

If the PrfOpenProfile function is successful, it returns a handle to the initialization file. Otherwise, it returns NULL, and the file is not created. Once you have an initialization-file handle, you can create new sections and settings in the file.

To close an initialization file, you use the PrfCloseProfile function.

Reading and Writing Settings

An application can store strings, integers, and binary data in an initialization file and retrieve them. To read from or write to an initialization file, your application must provide a section name and a key name that specify which setting to read or change. If the section or key name you specify in a writing operation does not exist in the file, it is added to the file and assigned the given value.

The following code fragment creates a section named "MyApp" and a key named "MainWindowColor" in a previously opened initialization file, and assigns the value of the RGB structure to the new setting:

```
HINI hini;
RGB rgb = { 0xff, 0x00, 0x00 };

PrfWriteProfileData(hini, "MyApp", "MainWindowColor", &rgb, sizeof(RGB));
```

To read a setting, your application can retrieve the size of the setting and then read the setting into an appropriate buffer by using the PrfQueryProfileSize and PrfQueryProfileData functions, as shown in the following example. This example reads the setting "MainWindowColor" from the "MyApp" section only if the size of the data is equal to the size of the RGB structure.

```
HINI hini;
ULONG cb;
RGB rgb;

PrfQueryProfileSize(hini, "MyApp", "MainWindowColor", &cb);
if (cb == sizeof(RGB))
    PrfQueryProfileData(hini, "MyApp", "MainWindowColor", &rgb, &cb);
```

An application can also read strings by using the PrfQueryProfileString function, write strings by using the PrfWriteProfileString function, and read integers (stored as strings) by using the PrfQueryProfileInt function.

Identifying the OS/2 Initialization Files

Your application can retrieve the names of the system initialization files by using the PrfQueryProfile function. Although the OS/2 initialization files are usually named os2.ini and os2sys.ini, you can use other files when starting the system.

The following example retrieves the names of the initialization files and copies their names to the strings szUserName and szSysName. Once you know the names of the OS/2 initialization files, you can use them to open the files and read settings.

```
CHAR szUserName[CCHMAXPATH];
CHAR szSysName[CCHMAXPATH];
HINI hini;

PRFPROFILE prfpro = { sizeof(szUserName), szUserName,
                      sizeof(szSysName), szSysName };

PrfQueryProfile(hini, &prfpro);
```

You can change the OS/2 initialization files to files of your choice by using the PrfReset function. This function requires the names of two initialization files and uses them as replacements for the os2.ini and os2sys.ini files. The system is then reset by using the settings in the new files.

Summary

Following are the OS/2 2.0 functions used with initialization files:

Table 36-1. Initialization File Functions

Function name	Description
PrfCloseProfile	Indicates that a profile is no longer available for use.
PrfOpenProfile	Indicates that a file is available for use as a profile
PrfQueryProfile	Returns a description of the current user and system profiles.
PrfQueryProfileData	Returns a string of binary data from the specified profile.
PrfQueryProfileInt	Returns an integer value from the specified profile.
PrfQueryProfileSize	Obtains the size, in bytes, of the value of a specified key for a specified application in the profile.
PrfQueryProfileString	Retrieves a string from the specified profile.
PrfReset	Defines which files are to be used as the user and system profiles.
PrfWriteProfileData	Writes a string of binary data into the specified profile.
PrfWriteProfileString	Writes a string of character data into the specified profile.

Appendix A. Comparison of 1989 and 1991 CUA User Interface Guidelines

Section	CUA Guidelines — 1989	CUA Guidelines — 1991
Accelerator	Accelerator term used.	Terminology change — called a Shortcut key.
Action bar	Action bar term used. Used if more than one action is available.	Terminology change — called a Menu bar. Used if more than six actions are available, or when any of the predefined menu bar actions are available.
Action message	Stop-sign symbol always used.	Question mark or stop-sign symbol may be used.
Audible feedback	Beep recommended.	Recommend using available audio capabilities as feedback.
Column heading	Alignment of columns and headings not addressed. Use of separators not addressed. Required headings not addressed.	Alignment of columns and headings are defined based on length. Recommend separators between columns and headings. Column headings not required if there is only one column.
Combination box	Default choices not addressed.	Recommend displaying a default choice.
Container	Addressed at a direction level only.	A new control. An object used to hold other objects.
Contents of menus	May contain action, routing, or settings (properties) choices. **Short menus** and **Full menus:** Not addressed.	May contain action or routing choices. Encourages using a notebook control for settings choices. **Short Menus** and **Full Menus** — the contents and techniques are defined.
Contextual help	Contextual help for direct-manipulation tasks not addressed.	Defined for direct-manipulation tasks.
Delete folder	Not addressed.	A container used to remove objects from the operating environment.
Dialog box	Dialog boxes used to continue users requests (movable, but not sizable).	Secondary windows used to continue users requests. Recommend they are movable and sizable. Terminology change - dialog box term no longer used.

Section	CUA Guidelines — 1989	CUA Guidelines — 1991
Direct manipulation	Direct manipulation discussed briefly.	Direct manipulation discussed as a pervasive technique.
		Recommendation to provide direct manipulation for all objects.
	Direct manipulation of split bar not addressed	Manipulation button drags split bar.
Do-not pointer	Not addressed.	Defines do-not pointer for use during direct-manipulation operations.
Drop-down combination box	Order and number of choices not addressed.	Recommend placing choices in numeric, alphabetic, or chronological order and display at least six choices in a box.
Drop-down list	Order and number of choices not addressed.	Recommend placing choices in numeric, alphabetic, or chronological order and display at least six choices in a box.
Edit menu	**Redo:** Not addressed.	**Redo** choice used to reverse the effect of an undo action.
	Create: Not addressed.	**Create** choice used to create a new object or a reflection of the current object using the clipboard.
	Find: Not addressed.	**Find** choice allows a user to search for an object or a part of an object.
Field prompts	Left-align field prompts only.	Allow left-aligned or right-aligned field prompts.
	Field prompts followed by colons shown in many examples.	Field prompts followed by colons no longer suggested or used in examples.
File menu	**File:** Name used for first menu choice on the menu bar.	File — used for application-oriented windows; "class name" used for first menu choice of object-oriented windows.
	Open as: Not addressed.	Opens another view of the object in another window.
	Print: Allows a window for more information	**Print:** Allows a window for more information, and allows a cascaded menu for printer selection
	Exit (optional)	Not used; performed by close action of system menu in associated primary window.
Folder	Not addressed.	System-provided container used to group objects.

Section	**CUA Guidelines — 1989**	**CUA Guidelines — 1991**
Group box	Capitalization rules not addressed. Not addressed.	Capitalize first letter only (some exceptions described). Recommend using only when white space or group headings would be insufficient.
Help menu	**Help** menu choices displayed with ellipses. **Help for help** choice **Extended help** choice **Keys help** Not addressed. **Help index** choice **About** choice — leads to a logo window.	**Help** menu choices are not displayed with ellipses. Terminology change — **Using help.** Position change in the **Help** menu. Terminology change — **General help** Removed from **Help** menu, now accessed from the help index. Recommend describing settings for buttons on pointing device in keys help. Position change in **Help** menu. Terminology change — **Product information** choice leads to a product-information window.
Hide	Not addressed.	A choice that removes a window and all associated windows from the workplace.
Hourglass pointer	Hourglass pointer term used.	Recommend displaying wait pointer over parts of a window. Terminology change — called a wait pointer. Two wait pointer visuals are available.
Information area	Not addressed.	Information area defined as part of a window where information appears about the object or choice that the cursor is on. Information about the normal completion of a process can also appear in the information area.
Information message	Used for normal processing situations when there are no additional actions available.	Used when additional information about a completed process is available and no progress indicator is displayed, or when a process cannot complete and there are no additional actions available.
In-use emphasis	Not addressed.	In-use emphasis defined for opened objects.

Section	CUA Guidelines — 1989	CUA Guidelines — 1991
Keyboard	Accelerator keys	Terminology change — Shortcut keys
	No guidance given about user changes.	If changed by users, changes reflected in menus and help.
	Case sensitivity not addressed.	Allow either upper or lowercase characters.
	Use of preferred modifiers not addressed.	Recommend using the Alt key element of shortcut key assignments to only provide access to mnemonics and to provide access to operating-environment-provided shortcut keys.
Message box	Special type of dialog box used for messages (modal and sizable)	Secondary windows used for messages. Recommend they are modeless and sizable.
		Terminology change — message box term no longer used.
Messages	Application name used for window title.	*Object name — action* used in window title.
	Messages are application modal and nonsizable.	Recommend to allow a user to continue interacting with parts of an object while message displayed and size messages.
	Controls in messages not addressed.	Recommend providing interactive controls in messages.
	Not addressed.	Describes displaying message symbol on icon if window is not open.
Modal and modeless	Modeless dialogs used only for repeat actions.	Modeless windows encouraged for all windows.
Mouse	Using mouse to create a reflection not addressed.	Ctrl+Shift+Manipulation button assigned to create reflection operation.
	Effect of move and copy operations on pointer visuals not addressed.	Move and copy operations effect on pointer visuals defined.
Multiple document interface	Used to view many objects or multiple views of same object. All windows contained within one window and share a menu bar.	Multiple windows used to view many objects or multiple views of the same object. Multiple document interface only addressed in the context of migration. Also see the **Windows** menu.
Notebook	Not addressed.	New control. Recommended for displaying settings and some types of objects.

Section	**CUA Guidelines — 1989**	**CUA Guidelines — 1991**
Options menu	Contains product-specific choices related to the application.	Used primarily in application-oriented windows. Encourages using a notebook control for these types of choices.
Pop-up menu	Not defined	Pop-up menus defined to display actions for indicated object. Shift+F10 and chording selection and manipulation buttons display pop-up menu of indicated object.
Progress indicator	Display a progress indicator for complex tasks. Not addressed. Only a **Stop** push button is defined for controlling the process. Title not addressed. Help not addressed. Removing the progress indicator not addressed.	Display a progress indicator for tasks that take more than 5 seconds. Display a progress indicator in action window where process is requested. **Stop, Pause,** and **Resume** push buttons defined for controlling the process. **Close** push button not allowed for stopping the process. Use the word "progress" in the window title. Recommend providing **Help.** Product removes the progress indicator if no special completion information needed; otherwise the user removes the progress indicator.
Pull-down menu	Recommended at least two choices in a pull-down menu.	Not addressed.

Section	CUA Guidelines — 1989	CUA Guidelines — 1991
Push button	Changing contents of a push button not addressed.	Use two push buttons, do not change content of same push button.
	Normal position is in lower area of window.	Place push buttons that affect an entire window horizontally at the bottom of the window, justified from the left edge. If a push button is associated with a component, place it near the component.
	Push buttons not allowed in windows with menu bars.	Push buttons allowed in windows with menu bars.
	Position of push buttons when sizing or scrolling not addressed.	Push buttons remain in same relative position when sizing or scrolling.
	Default push button required for each window containing push buttons.	Default push button recommended for each window containing push buttons.
	Pause, Resume, Close, and **Continue:** not addressed.	Recommended usage described for **Pause, Resume, Close,** and **Continue.**
Radio button	**None** choice not addressed.	Recommend **None** choice if a user can choose not to select any of a set of choices.
Reflection	Not addressed.	An object represented by more than one icon.
Restore of minimized windows	**Restore** returns to middle size.	**Restore** returns to previous size and position.
Scroll bar	Slider box — part of the scroll bar used to scroll.	Terminology change — scroll box.
Scroll increment	General descriptions given, text examples provided.	Recommendations included for icons, graphics, and text.
Secondary window	Term used only to refer to movable, sizable windows dependent on another primary window.	Terminology change—definition expanded to include all windows dependent on another primary window (independent of whether they are movable or sizable).
	May not be minimized	May be minimized when used to display views of objects.
Selected emphasis	Referred to as selected emphasis	Terminology change — selected-state emphasis.
	Use inverse color for selected emphasis on text.	For all objects show by changing the foreground and background colors.

Section	CUA Guidelines — 1989	CUA Guidelines — 1991
Selected menu	Functions were available in the **File** menu for list handlers.	New menu-bar choice used for actions on selected objects within the window.
	Open as choice — Not addressed.	Choice used to display another view of an object in a window.
Separators	Not addressed.	White space recommended except in menus.
Single-line entry field	Specific rules for visible length not addressed.	When the length of data is predictable, such as time or date, the field should be entirely visible.
Slider	Not addressed.	New control to represent a quantity and its relationship to a range of possible values.
	Usage of scroll bar for numeric values not addressed.	Slider control used.
Source emphasis and target emphasis	Not defined	Defines source emphasis and target emphasis for direct-manipulation operations.
Spin button	Not defined.	Order of choices is based on type of data.
Split window	Allows only one vertical and one horizontal split.	Allows multiple vertical and horizontal splits.
Status area	Not addressed.	Status area defined as part of a window where information appears about the state of an object or the state of a particular view of an object.
System menu	**Close** choice does not address saving window status information.	**Close** choice recommends saving window state, such as its position, size, and associated messages.
	Close choice only addressed for dialog boxes.	Result of **Close** choice defined depending on window content.
Title bar mini-icon	Introduced in the workplace environment and referred to as the Title bar mini-icon.	Referred to as the small icon in the title bar.
	Not addressed.	Defines use of target emphasis during direct-manipulation operations.
Tool palette	Briefly described	Content and usage described.

Section	CUA Guidelines — 1989	CUA Guidelines — 1991
View menu	Names of views addressed in the **View** menu.	Names of views are listed at the top of the **View** menu.
	All: Used to see the entire contents.	**Include:** Used to see the entire contents or part of the contents.
	Some: Used to see part of the contents.	**Include:** Used to see the entire contents or part of the contents.
	By: Used to sort the contents.	Terminology change — **Sort.**
	Refresh: Not addressed.	**Refresh → On/Off** used to allow a user to control updates to the window contents.
	Refresh now: Not addressed.	**Refresh now:** Used to update the window contents immediately.
Warning message	**Yes** and **No** push buttons allowed.	Recommend using **Continue** push buttons and action push buttons.
Window menu	Used for MDI windows.	Terminology change — **Windows** menu used to access and manage related windows.
Window title	"Application name — OS/2 file name"	Added window title rules for object-oriented windows
Work area	Not addressed.	A container used to group objects by task.

Appendix B. Documenting the CUA User Interface in Products

The following information is provided to help you document your product's user interface and associated information. The following table contains both technical and user terms. The user terms are defined and suggestions are given on how to explain the technical concepts to users.

General Terminology Guidelines

The terminology used in your product should be suited to the task domain of the product's users. For example, if the primary users of a product are programmers, use terms programmers are familiar with and understand; similarly, if the primary users are members of the medical community or the insurance community, use terms those users will expect and understand.

If your product has a particular implementation of a concept that you want to include in the definition, you may append that information to the end of the definition. Precede the appended information with a phrase such as: `In myproduct,....`

Predefined user-interface terminology (terms that appear in the table in **bold** text) must be used for all users. Synonyms for these terms are not allowed.

How to Use This Table

Use the terms and their definitions in your product documentation just as they appear in the following table. Some of the terms that appear as choices on the user interface can either be action or routing choices. If they are used by your product as routing choices, append either an ellipsis or a right-pointing arrow to the term as appropriate.

Some of the technical terms in this table do not have equivalent user terms. To help you explain to users the concepts represented by these technical terms, suggestions are given in the right-hand column of the table. The documentation suggestions appear in *italic* text to distinguish them from term definitions.

Other technical terms in the table have equivalent user terms; for example, look at the term "action message" in the table. In the right-hand column, you are referred to "message" for the definition; "message" is the user equivalent of "action message."

Note: Predefined capitalization rules have been applied to the user-interface terminology in the following table. Terms in **bold** text appear in CUA-conforming user interfaces as choices in menus, labels on push buttons, and labels associated with icons.

Table B-1 (Page 1 of 14). Technical Terms with Equivalent User Terms and User Definitions		
Technical Term	**User Term**	**User Definition or Documentation Suggestion**
action	action	An action performs a task on an object. A user requests actions by selecting a choice from a menu, interacting with buttons in a window, or by manipulating objects directly.
action message		See *message.*
active window	active window	The window that can receive input from the keyboard. It is distinguishable by the unique color of its title bar and window border.
Apply	**Apply**	A push button that carries out the selected choices in a window without closing the window.
audible feedback		*Use "beep" or describe the sound.*
automatic selection		*A selection technique in which moving the keyboard cursor automatically changes the current selection. A user does not have to identify a choice or object to select it, selection occurs automatically as the cursor moves among the choices or objects.*
border	border	A visual indication of the boundaries of a window.
button	button	(1) A mechanism on a pointing device, such as a mouse, used to request or initiate an action or a process. (2) A graphical device that identifies a choice. (3) A graphical mechanism that, when selected, performs a visible action. For example, when a user clicks on a list button, a list of choices appears.
Cancel	**Cancel**	A push button that removes a window without applying any changes made in that window.
cascaded menu	cascaded menu	A menu that appears from, and contains choices related to, a cascading choice in another menu.
cascading choice	cascading choice	A choice on a menu that, when selected, presents another menu with additional related choices.
check box	check box	A square box with associated text that represents a choice. When a user selects the choice, the check box is filled to indicate that the choice is selected. The user can clear the check box by selecting the choice again, thereby deselecting the choice.

Table B-1 (Page 2 of 14). Technical Terms with Equivalent User Terms and User Definitions

Technical Term	**User Term**	**User Definition or Documentation Suggestion**
check mark	check mark	A character (√) that indicates that a choice is active.
choice	choice	Graphics or text that a user can select to modify or manipulate an object. Choices appear in menus, on push buttons, and in fields as in, for example, a field of radio buttons.
chord	chord	To press more than one button on a pointing device while the pointer is within the limits that the user has specified for the operating environment.
Clear	**Clear**	A choice that removes a selected object and leaves the visible space that it occupied.
click	click	To press and release a button on a pointing device without moving the pointer off of the object or choice.
clipboard	clipboard	An area of storage provided by the system to hold data temporarily.
Close	**Close**	A choice that removes a window and all of the windows associated with it from the workplace. For example, if a user is performs a task in a window and a message appears, or the user asks for help, both the message and the help windows disappear when the user closes the original window.
combination box		*Refer to the list of objects or choices that a user can access by selecting the list button, and the entry field into which a user can type directly.*
container	container	A visual user-interface component whose specific purpose is to hold objects.
contextual help	contextual help	Help information about the specific choice or object that the cursor is on. The help is contextual because it provides information about the item in its current context.
control		*Name the control if it is a user term; otherwise describe it, its various parts, or tell the user how to interact with it.*
Copy	**Copy**	A choice that places a copy of a selected object onto the clipboard.
Create	**Create**	An action choice that produces a new object, similar to a selected object, and places it on the clipboard.

Table B-1 (Page 3 of 14). Technical Terms with Equivalent User Terms and User Definitions		
Technical Term	**User Term**	**User Definition or Documentation Suggestion**
current-setting indicator		*A mark, such as a checkmark, an "X" in a check box, or a filled circle in a radio button, that indicates that a choice is currently selected.*
current state		*The state of an object or choice, active or inactive, that allows it to be selected or directly manipulated.*
cursor	cursor	A visible indication of the position where user interaction with the keyboard will appear. The keyboard cursors are the selection cursor and the text cursor.
Cut	**Cut**	A choice that moves a selected object and places it onto the clipboard. The space it occupied is usually filled by the remaining object or objects in the window.
data transfer		*The movement of data from one object to another by way of the clipboard or by direct manipulation*
Delete	**Delete**	A choice that removes a selected object. The space it occupied is usually filled by the remaining object or objects in the window.
delete folder	delete folder	A folder that holds objects and that will remove the objects it holds from a user's system. A delete folder could delete objects immediately, or it could allow the user to specify when the objects are to be deleted.
Deselect all	**Deselect all**	A choice that cancels the selection of all of the objects that have been selected in that window.
default action		*Explain to the user that when some action is taken, such as pressing the Enter key, the default action (describe the emphasis that identifies it) will be performed.*
descriptive text		*Text used in addition to a field prompt to give more information about a field.*
detent		*A point on a slider that represents an exact value to which a user can move the slider arm.*
dialog	dialog	The interaction between a user and a computer.
dimmed		*Reduced contrast that indicates that a choice or object cannot be selected or directly manipulated.*

Table B-1 (Page 4 of 14). Technical Terms with Equivalent User Terms and User Definitions

Technical Term	**User Term**	**User Definition or Documentation Suggestion**
direct manipulation	direct manipulation	Techniques that a user employs to work with objects directly, through a pointing device, or through the objects' context menus.
directory	directory	A container of files and other directories.
double-click	double-click	To press and release a button on a pointing device twice while a pointer is within the limits that the user has specified for the operating environment.
drag	drag	To use a pointing device to move an object. For example, a user can drag a window border to make it larger.
drag and drop	drag and drop	To directly manipulate an object by moving it and placing it somewhere else using a pointing device.
drop-down combination box		*Tell the user how to interact with it; refer to the entry field and the list button.*
drop-down list		*Tell the user how to interact with it; refer to the list of items that are shown when the user clicks on the list button.*
Edit	**Edit**	A choice on a menu bar that provides access to other choices that enable a user to modify data.
emphasis	emphasis	Highlighting, color change, or other visible indication of the condition of an object or choice and the effect of that condition on a user's ability to interact with that object or choice. Emphasis can also give a user additional information about the state of an object or choice. **Note:** Describe to the user what the emphasis indicates. For example, that selected-state emphasis shows that a choice or object is selected.
entry field	entry field	An area into which a user types or places text. Its boundaries are usually indicated.
extended selection		*A type of selection usually used for the selection of a single object. A user can extend selection to more than one object, if required.*
field	field	An identifiable area in a window. Examples of fields are: an entry field, into which a user can type or place text, and a field of radio button choices, from which a user can select one choice.
field prompt	field prompt	Text that identifies a field, such as an entry field or a field of check boxes.

Table B-1 (Page 5 of 14). Technical Terms with Equivalent User Terms and User Definitions		
Technical Term	**User Term**	**User Definition or Documentation Suggestion**
File	**File**	A choice on a menu bar that provides access to other choices that enable a user to work with the object in the window as a whole.
Find	**Find**	A choice or push button that initiates a search for an object or within an object displayed in that window. A user can specify the criteria to be used for the search.
first-letter navigation		*A navigation and selection technique in which users select a choice in a list by typing the first character of the choice they want to select*
folder	folder	A container used to organize objects.
Full menus	**Full menus**	A choice that a user selects to see all of the choices available in menus.
General help	**General help**	A choice that gives a user a brief overview of each action or task, or both, that a user can perform within a window.
group heading	group heading	A heading that identifies a set of related fields.
Help	**Help**	A choice that gives a user access to helpful information about objects, choices, tasks, and products. A **Help** choice can appear on a menu bar or as a push button.
Help index	**Help index**	A choice on the **Help** menu that presents an alphabetic listing of help topics for an object or a product.
Hide	**Hide**	A choice that removes a window and all associated windows from the workplace.
I-beam pointer	I-beam pointer	A pointer that indicates that the pointer is over an area that can be edited, for example, an entry field.
icon	icon	A graphical representation of an object, consisting of an image, image background, and a label.
inactive window	inactive window	A window that is not receiving keyboard input. It can be distinguished from an active window by the difference in its title bar and border colors.
Include	**Include**	A choice that presents a window in which a user can specify a reduced or expanded set of objects, so that only the objects included in the reduced or expanded set are displayed.

Table B-1 (Page 6 of 14). Technical Terms with Equivalent User Terms and User Definitions		
Technical Term	**User Term**	**User Definition or Documentation Suggestion**
information area	information area	A specific part of a window in which information about the object or choice that the cursor is on is displayed. The information area can also contain a message about the completion of a process.
information message		See *message*.
initial value		*Information that appears in an entry field when that entry field is first displayed*
input focus		*The position, indicated on the screen, where a user's interaction with the keyboard will appear.*
in-use emphasis		See *emphasis*.
Keys help	**Keys help**	A choice that presents a listing of all the key assignments for an object or a product.
list box		*A control that contains a list of objects or settings choices that a user can select from.*
list button	list button	A button labeled with an underlined down-arrow that presents a list of valid objects or choices that can be selected for that field.
manipulation button	manipulation button	The button on a pointing device a user presses to directly manipulate an object, for example mouse button 2 is the default manipulation button on a two-button mouse.
marquee box		*The rectangle that appears during a selection technique in which a user selects objects by drawing a box around them with a pointing device.*
marquee selection		*A technique that a user employs to select objects by using a pointing device to draw a box around them.*
Maximize	**Maximize**	A choice that enlarges a window to its largest possible size.
maximize button	maximize button	A button in the rightmost part of a title bar that a user clicks on to enlarge the window to its largest possible size.
menu	menu	A list of choices that can be applied to an object. A menu can contain choices that are not available for selection in certain contexts. Those choices are indicated by reduced contrast.

Table B-1 (Page 7 of 14). Technical Terms with Equivalent User Terms and User Definitions		
Technical Term	**User Term**	**User Definition or Documentation Suggestion**
menu bar	menu bar	The area near the top of a window, below the title bar and above the rest of the window, that contains choices that provide access to other menus.
menu-bar choice	menu-bar choice	A graphical or textual item on a menu bar, which provides access to menus that contain choices that can be applied to an object.
menu button	menu button	The button on a pointing device that a user presses to view a pop-up menu associated with an object, for example mouse button 3 is the default menu button on a three-button mouse.
menu choice	menu choice	A graphical or textual item on a menu. A user selects a menu choice to work with an object in some way.
message	message	Information not requested by a user but displayed by a product in response to an unexpected event or when something undesirable could occur.
Minimize	**Minimize**	A choice that reduces a window to its smallest possible size and removes all of the windows associated with that window from the screen.
minimize button	minimize button	A button, located next to the rightmost button in a title bar, that reduces the window to its smallest possible size and removes all the windows associated with that window from the screen.
mnemonic		*A selection technique; refer to the "underlined character" or the "character in parentheses" that a user can type to move the cursor to a choice or to select the choice that the cursor is on.*
mouse	mouse	A commonly used pointing device, containing one or more buttons, with which a user can interact with a product or the operating environment.
mouse button	mouse button	A mechanism on a mouse pointing device used to select objects or choices, initiate actions, or directly manipulate objects. that a user presses to interact with a computer system. The button makes a "clicking" sound when pressed and released.
Move	**Move**	A choice that moves a window to a different location on the work area.
multiple-line entry field	entry field	See *entry field.*

Table B-1 (Page 8 of 14). Technical Terms with Equivalent User Terms and User Definitions

Technical Term	**User Term**	**User Definition or Documentation Suggestion**
New	**New**	A choice that creates another object from an existing object. The new object will appear in the existing window.
notebook	notebook	A graphical representation that resembles a spiral-bound notebook that contains pages separated into sections by tabbed divider-pages. A user can turn the pages of a notebook to move from one section to another.
object	object	An item that a user can manipulate as a single unit to perform a task. An object can appear as text, an icon, or both.
Off	**Off**	A choice that appears in the cascaded menu from the **Refresh** choice. It sets the refresh function to off.
OK	**OK**	A push button that accepts the information in a window and closes it. If the window contains changed information, those changes are applied before the window is closed.
On	**On**	A choice that appears in a cascaded menu from the **Refresh** choice. It immediately refreshes the view in a window.
Open	**Open**	A choice that leads to a window in which users can select the object they want to open.
Open as	**Open as**	A cascading choice that leads to a cascaded menu which contains choices that a user can select to determine how an object is presented.
Options	**Options**	A choice on a menu bar that provides access to other choices that enable a user to customize a product or application.
palette	palette	A set of mutually exclusive, typically graphical, choices.
pane	pane	One of the separate areas in a split window.
Paste	**Paste**	A choice that places the contents of the clipboard at the current cursor position.
pointer	pointer	A symbol, usually in the shape of an arrow, that a user can move with a pointing device. Users place the pointer over objects they want to work with.
pointing device	pointing device	A device, such as a mouse, trackball, or joystick, used to move a pointer on the screen.

Table B-1 (Page 9 of 14). Technical Terms with Equivalent User Terms and User Definitions

Technical Term	User Term	User Definition or Documentation Suggestion
point selection	point selection	*A selection technique in which a user selects or deselects an item by clicking the selection button on a mouse while the pointer is positioned over an object or choice.*
pop-up menu	pop-up menu	A menu that, when requested, appears next to the object it is associated with.
primary window		See *window*.
Print	**Print**	A choice that prepares and schedules an object to be printed on a designated printer.
Product Information	**Product Information**	A choice that displays a window that contains information about an application or product, such as its copyright notice, a logo, or both.
progress indicator	progress indicator	Visual user-interface components that inform a user about the status of a computer process.
pull-down menu		See *menu*.
push button	push button	A button, labeled with text, graphics, or both, that represents an action that will be initiated when a user selects it.
radio button	radio button	A circle with text beside it. Radio buttons are combined to show a user a fixed set of choices from which the user can select one. The circle becomes partially filled when a choice is selected.
random-point selection		*A selection technique in which a user presses a mouse button and holds it down while moving the pointer so that the pointer travels to a different location on the screen. Everything the pointer touches while the button is held down is selected. Random-point selection ends when the mouse button is released.*
range selection		*A technique in which a user selects multiple objects in a range by identifying a beginning and end corner. When the second corner is identified, all objects within the specified range are selected.*
range-swipe selection		*A selection technique in which a user moves a pointer across a range of objects. Each object becomes selected as the pointer touches it.*

Table B-1 (Page 10 of 14). Technical Terms with Equivalent User Terms and User Definitions

Technical Term	User Term	User Definition or Documentation Suggestion
Redo	**Redo**	A choice that reverses the effect of the most recently performed undo operation on an object, returning the object to the state it was in before the undo operation was performed.
reflection		*An object that is represented by more than one icon.*
Refresh	**Refresh**	A cascading choice that gives a user access to other choices (**On** and **Off**) that control whether changes made to underlying data in a window are displayed immediately, not displayed at all, or displayed at a later time.
Refresh now	**Refresh now**	A choice that shows changes made to underlying data in a window immediately.
Reset	**Reset**	A push button that returns an object to the condition it was in when it was last opened, or to the condition it was in before the most recent changes were applied to it.
Restore	**Restore**	A choice that returns a window to the size it was and the position it was in before the user minimized or maximized the window.
restore button	restore button	A button that appears in the rightmost corner of the title bar after a window has been maximized. When the restore button is selected, the window returns to the size it was before it was maximized.
Retry	**Retry**	A push button that, when selected, attempts to complete an interrupted process.
Save	**Save**	A choice that stores an object onto a storage device, such as a disk or diskette.
Save as	**Save as**	A choice that creates a new object from an existing object and leaves the existing object as it was.
screen	screen	The physical surface of a display device upon which information is shown to users.
scrollable entry field		*An entry field that can be scrolled.*
scroll bar	scroll bar	A window component that shows a user that more information is available in a particular direction and can be scrolled into view. Scroll bars can be either horizontal or vertical.

Table B-1 (Page 11 of 14). Technical Terms with Equivalent User Terms and User Definitions

Technical Term	User Term	User Definition or Documentation Suggestion
scroll box	scroll box	The part of a scroll bar that indicates the position of the visible information relative to the total amount of information available in a window. A user clicks on a scroll box with a pointing device and manipulates it to see information that is not currently visible.
scrolling increment		*A fixed amount of information that can be scrolled with a single scrolling action.*
secondary window		See *window.*
select	select	To explicitly identify one or more objects to which a subsequent choice will apply.
Select all	**Select all**	A choice that causes all of the objects in a window to be selected.
Selected	**Selected**	A choice in the menu bar that provides access to choices that apply to the selected objects in the current view. Products can change the name of the choice to match the types of objects that appear in the current view, for example if a view contains only document objects, a product might name this choice **Documents.**
selected-state emphasis		See *emphasis.*
selection	selection	The process of explicitly identifying one or more objects to which a subsequent choice will apply.
selection button	selection button	The button on a pointing device that a user presses to select an object, for example mouse button 1 is the select button on a two-button mouse.
selection cursor	selection cursor	A keyboard cursor, in the shape of a dotted outline box, that moves as users indicate the choice they want to interact with.
Settings	**Settings**	A choice that sets characteristics of objects or displays identifying characteristics of objects.
shortcut key	shortcut key	A key or combination of keys assigned to a menu choice that initiates that choice, even if the associated menu is not currently displayed.
Short menus	**Short menus**	A choice that reduces the number of choices that appear in menus.
single-line entry field		See *entry field.*

Table B-1 (Page 12 of 14). Technical Terms with Equivalent User Terms and User Definitions

Technical Term	User Term	User Definition or Documentation Suggestion
Size	**Size**	An action choice that allows a user to change the size of a window.
slider	slider	A visual component of a user interface that represents a quantity and its relationship to the range of possible values for that quantity. A user can also change the value of the quantity.
slider arm	slider arm	The visual indicator in the slider that a user can move to change the numerical value.
slider button	slider button	A button on a slider that a user clicks on to move the slider arm one increment in a particular direction, as indicated by the directional arrow on the button.
slider shaft		*The part of the slider on which the slider arm moves.*
Sort	**Sort**	A choice that arranges the objects in a view into a specified order.
source emphasis		See *emphasis.*
spin button	spin button	A component used to display, in sequence, a ring of related but mutually exclusive choices. A user can accept the value displayed in the entry field or can type a valid choice into the entry field.
split box	split box	A box in the scroll bar of a window that a user can interact with to split a window into separate panes.
Split	**Split**	A choice that divides a window into more than one pane. Also, a choice used to change the size of each pane.
status area	status area	A part of a window where information appears that shows the state of an object or the state of a particular view of an object.
system menu	system menu	A menu that appears from the system menu symbol in the leftmost part of a title bar. It contains choices that affect the window or the view it contains.
system-menu symbol	system-menu symbol	A symbol (shaped like a spacebar) in the leftmost corner of a title bar that gives a user access to choices that affect the window or the view it contains.
tabbed divider-page	tabbed divider-page	A graphical representation of a tabbed page in a notebook. Tabbed divider-pages separate sections of the notebook.

Table B-1 (Page 13 of 14). Technical Terms with Equivalent User Terms and User Definitions

Technical Term	User Term	User Definition or Documentation Suggestion
table	table	An object, such as a spreadsheet, that is organized in a grid of rows and columns. Each intersection is called a cell and can contain objects, such as text or graphics, or both.
target emphasis		See *emphasis.*
text cursor	text cursor	A symbol displayed in text that shows a user where typed input will appear.
title bar	title bar	The area at the top of each window that contains the system menu symbol, a small icon, a window title, and the maximize, minimize, and restore buttons.
tool palette	tool palette	A palette whose choices represent tools. When a user selects a choice from the tool palette and moves the pointer into the window, the pointer changes to the shape of the selected choice and the pointing device performs the operation indicated by the pointer. For example, a user might select a "pencil" choice from the tool palette to make a drawing in the window.
Tutorial	**Tutorial**	A choice that gives a user access to online educational information.
unavailable-state emphasis		See *emphasis.*
Undo	**Undo**	A choice that reverses the effect of the most recently performed operation on an object, returning the object to the state it was in before the operation was performed.
Using help	**Using help**	A choice on the **Help** menu that gives a user information about how the help function works.
value set		*A set of mutually exclusive, graphical or textual choices.*
View	**View**	A choice on a menu bar that provides access to other choices that enable a user to choose how an object is presented, how much information is presented, what order it is presented in, and other choices related to the way an object is presented.
visible cue		*Describe the visual cue and tell the user what it indicates.*

Table B-1 (Page 14 of 14). Technical Terms with Equivalent User Terms and User Definitions

Technical Term	User Term	User Definition or Documentation Suggestion
wait pointer		*A pointer that indicates that the computer is performing a process and that the user cannot interact with the part of the underlying window that the wait pointer is positioned over.*
window	window	An area with visible boundaries that presents a view of an object or with which a user conducts a dialog with a computer system.
Windows	**Windows**	A choice on a menu bar that provides access to other choices with which users can manage all of the open windows on their system that are associated with the product.
Window list	**Window list**	A choice that presents a list of all of the open windows associated with the window from which the **Window list** choice was selected.
window title	window title	The area on a title bar that contains a short description of the contents of the window.
work area	work area	A container used to group windows and objects to perform a task. Users can modify sample work areas to suit their own needs.
workplace	workplace	A container that fills the entire screen and holds all of the objects that make up the user interface.

Appendix C. List of Approved Deviations from CUA User Interface Guidelines

Table C-1 (Page 1 of 5). CUA-Approved Deviations and Guidelines

Deviation	Fundamental Compliance Guideline
TUTORIAL — Keyboard support is not provided for the user to access the push buttons.	Provide access to all functions of an object using equivalent (although not necessarily identical) keyboard and pointing-device techniques.
TUTORIAL — Tab key moves the cursor within the value set field.	Tab key moves the cursor to the next field.
TUTORIAL — Emphasis is not shown on the default push button.	Provide a visual cue (i.e.dark border) to indicate which push button in a window performs the default action for that window.
TUTORIAL — **Exit** push button performs the **Close** function.	Use predefined label for each predefined choice.
EDIT FONT action window — Pressing Enter does not cause the default action to begin.	Pressing the Enter key or double clicking the selection button while the pointer is on an object or choice performs the default action or choice.
NOTEBOOK — The cursor is not visible on the notebook page when the keyboard is used to move the focus from a tab to the page.	Display a cursor to indicate the current position of the keyboard-input focus.
NOTEBOOK — Up arrow key moves the cursor from the notebook page to a notebook tab.	Alt+Up arrow moves the cursor from a notebook page to a notebook tab or page push button.
DESKTOP — Keyboard support is not provided for the user to reposition objects on the Desktop.	Provide access to all functions of an object using equivalent (although not necessarily identical) keyboard and pointing device techniques.
DESKTOP — Shift+F10 displays the Desktop pop-up menu instead of the pop-up menu for the object on which the cursor is positioned.	If pop-up menus are provided, enable a user to display the pop-up menu using the keyboard by pressing Shift+F10 when the cursor is on the object.
DESKTOP — Pop-up menu cannot be obtained via the keyboard while objects are selected on the Desktop.	Alt+Up arrow, followed by Shift+F10, displays the Desktop pop-up menu.
SYSTEM ERROR message does not have a system menu.	Provide a system menu for each window.
SHREDDER — Mnemonic is missing from the **Refresh** choice in the pop-up menu.	Assign **R** as the mnemonic for the **Refresh** choice.
COPY, MOVE, and CREATE SHADOW windows — Tab key moves the cursor from a notebook page to the next control.	Ctrl+Tab key moves the cursor to the next control when the cursor is in a notebook.

Table C-1 (Page 2 of 5). CUA-Approved Deviations and Guidelines	
Deviation	**Fundamental Compliance Guideline**
Alt+Tab key switches between unassociated windows.	Alt+Esc is the assigned key combination to switch from a window to an unassociated primary window.
GLOSSARY LIST window — Mnemonic is missing from the **Search** push button.	Assign a unique mnemonic to each textual push-button choice that does not have a specific keyboard access mechanism, such as **Esc** for the **Cancel** push button or **F1** for the **Help** push button, unless no meaningful unique mnemonic can be found.
GLOSSARY LIST window — Push buttons are not left-justified.	Push buttons that affect the entire window should be placed horizontally, at the bottom of the window, left-justified.
GLOSSARY SETTINGS window — Mnemonic is missing from the **Undo** push button.	Assign a unique mnemonic to each textual push button choice that does not have a specific keyboard access mechanism, such as **Esc** for the **Cancel** push button or **F1** for the **Help** push button, unless no meaningful unique mnemonic can be found.
GLOSSARY SETTINGS window — On the Properties page, the Tab key moves the cursor within the push-button field.	The Tab key moves the cursor to the next field.
MASTER INDEX SETTINGS window — On the Properties page, the mnemonic is missing from the **Undo** push button.	Assign a unique mnemonic to each textual push button choice that does not have a specific keyboard access mechanism, such as **Esc** for the **Cancel** push button or **F1** for the **Help** push button, unless no meaningful unique mnemonic can be found.
DIALOG EDITOR — The **Help** menu choice and all the choices on the pull-down menu are displayed with unavailable-state emphasis.	Do not display unavailable-state emphasis on routing choices that lead to pull-down menus or cascaded menus. If a choice is never available to a particular user, do not display it in a menu, and do not save space for it in a menu.
FONT EDITOR — The **Help** menu choice and all the choices on the pull-down menu are displayed with unavailable-state emphasis.	Do not display unavailable-state emphasis on routing choices that lead to pull-down menus or cascaded menus. If a choice is never available to a particular user, do not display it in a menu, and do not save space for it in a menu.
FORMAT — In the Progress window, the mnemonic is missing from the **Stop** push button.	Provide a predefined mnemonic for each predefined choice. Assign **S** as the mnemonic for the **Stop** push button.
FORMAT — In Progress window, the **Close** push button is missing.	Provide a push button that enables the user to close the progress indicator window without affecting the process.
MOUSE SETTINGS — Arrow key moves the cursor between the radio button field and the checkbox field.	Arrow keys move the cursor in the direction of the arrow shown on each arrow key.
SYSTEM SETTINGS — Tab key moves the cursor within the checkbox field.	The Tab key moves the cursor to the next field.

Table C-1 (Page 3 of 5). CUA-Approved Deviations and Guidelines	
Deviation	**Fundamental Compliance Guideline**
MENU SETTINGS window — Incorrect terminology and no mnemonic for the predefined push button are used.	Assign **O** and the mnemonic for the **OK** push button. Use the predefined label **OK** instead of **Ok** for the predefined choice.
POP-UP MENU for an open window — The **Close** choice is presented in the first-level menu and in the cascaded menu for the **Window** choice.	If the **Close** choice is provided, place it on the system menu only or on both the system menu and a push button in the window.
POP-UP MENU for an open window — Mnemonic is missing from the **Close** choice in the first-level menu.	Assign **C** as the mnemonic for the **Close** choice.
CLIPBOARD VIEWER — Tab key moves the cursor within the push button field (for example, in the "Render format" window).	Tab key moves the cursor to the next field.
CLIPBOARD VIEWER — Unavailable-state emphasis is shown on the menu bar choices (for example, the **Display** menu).	Do not display unavailable-state emphasis on routing choices that lead to pull-down menus or cascaded menus.
CLIPBOARD VIEWER — **Exit** must be removed from the **File** menu.	Use the predefined label for each predefined choice.
CLIPBOARD VIEWER — In the **Field** menu, the **Import** and **Export** choices are never available, yet they are displayed with unavailable-state emphasis.	If a choice is never available to a particular user, do not display the choice instead of displaying it with unavailable-state emphasis.
System configuration window — Push buttons are not left-justified.	Push buttons that affect the entire window must be placed horizontally, at the bottom of the window, left-justified.
OS/2 SETUP and INSTALLATION — In the warning message, the mnemonic is missing from the **OK** push button.	Assign **O** as the mnemonic for the **OK** push button.
FONT PALETTE — Target emphasis is not displayed during a direct manipulation operation.	Display target emphasis during a direct manipulation operation when the hot spot of the pointer is over an object that supports direct manipulation.
FONT PALETTE — Tab key moves the cursor within a push button field.	Tab key moves the cursor to the next field.
MASTER INDEX — Tab key moves the cursor from a notebook page to the notebook tab.	Alt+ Up arrow moves the cursor from a notebook page to a notebook tab or page push button.

Table C-1 (Page 4 of 5). CUA-Approved Deviations and Guidelines

Deviation	**Fundamental Compliance Guideline**
DOS SETTINGS — Mnemonics are assigned to **Help** and **Cancel** push buttons.	Assign a unique mnemonic to each textual push button choice that does not have a specific keyboard access mechanism, such as **Esc** for the **Cancel** push button or **F1** for the **Help** push button, unless no unique mnemonic can be found.
DEVICE DRIVE INSTALL — **Exit** push button performs **Close** function.	Use predefined label for each predefined choice.
ICON EDITOR — Change **Edit** push button to **Open** and remove ellipsis (...).	Use predefined label for each predefined choice. When a push button is used as a routing choice, use an ellipsis following the choice text.
CASCADE MENUS — Selecting a cascading choice for a conditional cascaded menu does not display the cascaded menu associated with that choice.	When a user selects a cascading choice, display the cascaded menu associated with that choice.
ICON/WINDOW TITLE — Direct editing of icon and window title is initiated by clicking mouse button 1 while holding down the Alt key.	Direct editing is initiated by point selection. The point selection function is assigned to a single click of mouse button 1 (selection button).
DIRECT MANIPULATION — Window sizing and movement is performed using mouse button 1 (selection button).	Direct manipulation is assigned to mouse button 2 for a two-button mouse.
DIRECT MANIPULATION — Pressing mouse button 2 while the pointer is on a template icon, then moving the mouse while holding down the mouse button, performs a *Create-on-drag*.	Ctrl + Manipulation button causes **Create** when *create-on-drag* is on.
DIRECT MANIPULATION — Objects do not consistently drop where a user releases mouse button 2.	Place source object at target position.
SCROLL BARS — Scroll bars are displayed only when information is not fully visible.	If information in the window is extendable, but is not currently scrollable, display the scroll bar with unavailable-state emphasis.
MENU BARS — No menu bars are provided on any object container windows, which also lack push buttons.	If a menu bar is not provided in a window displaying a view of an object, place all action and routing choices on push buttons in that window, except for those choices that appear on the system menu.
SYSTEM MENU — Object and system menu functions are on one pull-down on OS/2 object container windows.	Provide a system menu for each window.

Deviation	Fundamental Compliance Guideline
POP-UP MENU DISPLAY — Pressing mouse button 2 while a pointer is over an object displays a pop-up menu for that object.	If pop-up menus are provided, enable a user to display the pop-up menu using a 2-button mouse, chording the mouse selection and manipulation buttons when the pointer is over the object.
POP-UP MENU — **Open** choice performs the **Open as** function.	Use the predefined label for each predefined choice.
POP-UP MENU — A selected object becomes deselected when the pop-up menu is canceled.	Do not change the state of a window or object when a pop-up menu is displayed. For example, do not change the selection state of any object.
CLIPBOARD — Objects cannot be cut and pasted on the Workplace.	Provide access to the clipboard for all objects that support data transfer.
VIEWS — User cannot change the view in an object window. All views are nested under **Open**, and the user must open another window onto the object to look at an alternative view.	Provide a **View** choice on the menu bar of each window that provides a menu bar when more than one view is available for an object or any of the following choices are provided: **Sort, Include, Refresh**, or **Refresh now**.
REVERSI — **Exit** in the Game pull-down performs the function assigned to **Close**.	Use the predefined label for each predefined choice.

Table C-1 (Page 5 of 5). CUA-Approved Deviations and Guidelines

Index

A

C

E

F

I

J

K

L

M

N

O

T

U

X

Y

Z

Special Characters

International Business
Machines Corporation

Printed in the
United States of America

10G6494

S10G-6494-00

P10G6494